Clube Delphi

Edição 169 :: R$ 14,90

DEVMEDIA

ListView: Listagem dinâmica em aplicações mobile
Manipule o componente via linha de código

Técnicas de debug no Delphi
Identifique e corrija erros mais rapidamente

APLICAÇÕES COM MONGODB NO DELPHI

Acessando bancos NoSQL com FireDAC

ISSN 1517990-7

MVP

R$ 1.000.000,00
INVESTIDOS EM CONTEÚDO NOS ÚLTIMOS 12 MESES.

APLIQUE ESSE INVESTIMENTO NA SUA CARREIRA...

E MOSTRE AO MERCADO QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's consumido + de **500.000** vezes

POR APENAS
R$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

DEVMEDIA

Clube Delphi

169ª Edição 2016 ISSN 1517990-7 Impresso no Brasil

Editor Geral

Joel Rodrigues (joelrlneto@gmail.com)

Jornalista Responsável

Kaline Dolabella - JP24185

Revisora Técnica

Anny Caroline (annycarolinegnr@gmail.com)

Capa e Diagramação

Romulo Araujo

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia: **www.devmedia.com.br/mvp**

DEVMEDIA

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Joel Rodrigues - Editor da Revista
joelrlneto@gmail.com

Joel Rodrigues - Editor Geral

Editor da ClubeDelphi Magazine

Sumário

04 – ListView: Listagem de dados dinâmica em aplicações mobile
[*Jones Granatyr e Jean Paul Barddal*]

Destaque – Boas Práticas

13 – Técnicas de debug no Delphi
[*Gutierry Antonio*]

Artigo no estilo Solução Completa

24 – Desenvolva aplicações com MongoDB no Delphi
[*Guilherme Lourenção Tempesta*]

ListView: Listagem de dados dinâmica em aplicações mobile

Carregue dados para o ListView de forma dinâmica via código

Recentemente, a demanda por aplicativos móveis no mercado comercial tem crescido cada vez mais, fato que torna o conhecimento dessa área indispensável para os desenvolvedores. Dentro desse contexto, o RAD Studio, juntamente com o FireMonkey, apresenta uma gama muito grande e diversificada de recursos para o desenvolvimento de tais aplicativos.

Nesse âmbito, um dos principais e mais utilizados componentes é o ListView, que realiza a exibição de dados apresentando-os no formato de linhas e colunas. Esse componente é passível de vários tipos de personalizações no que tange à apresentação da lista dos itens, por meio de mudanças nas características de aparência, tais como acréscimo de legendas ou de detalhes em um texto, associação de imagens ou ainda a adição de botões ou outros tipos de componentes associados a cada registro. Outro aspecto importante do ListView é o suporte apropriado ao scroll, o que permite a criação de uma lista contendo vários itens e isenta o desenvolvedor da preocupação se o conteúdo caberá em um dispositivo de dimensões reduzidas.

Em suma, o ListView é um componente otimizado para apresentação de dados em dispositivos móveis. Com isso ele se torna um substituto natural do componente TDBGrid, largamente utilizado desde as primeiras versões do Delphi em aplicações desktop para apresentação de dados. Essa migração do DBGrid para o ListView é um processo que vem ocorrendo nas últimas versões do Delphi. Ao criar uma nova aplicação multidispositivo utilizando o FireMonkey, por exemplo, o componente DBGrid não se encontra mais disponível.

Fique por Dentro

Para o desenvolvimento de aplicativos móveis, o ListView é considerado o principal componente do FireMonkey para a listagem de registros, possibilitando uma gama de recursos visuais que podem agilizar muito o desenvolvimento dos aplicativos, por exemplo os Live Bindings, usados para a vinculação desse componente com recursos de acesso a dados. Os dados para popular um ListView podem ser carregados de diversas fontes, tais como bases de dados, dataSets em memória ou dados extraídos da web. Com isso, o poder visual do RAD Studio pode não ser suficiente, fazendo-se necessário o conhecimento de como manipular esses componentes utilizando linhas de código e preenchimento dinâmico de uma lista de registros.

Alguns dos recursos visuais que o RAD Studio fornece, como a vinculação automática de dados feita por meio de LiveBindings, podem não ser adequados em todas as situações. Isso ocorre porque em algumas situações bem específicas os componentes de acesso a dados do IDE (ClientDataSet ou queries, por exemplo) podem não atender a certos requisitos do cenário a ser trabalhado. Um exemplo disso é o acesso remoto a dados localizados nas nuvens ou até mesmo o acesso a um arquivo de texto local, situações em que o formato dos dados pode ser diferente da estrutura exigida por um componente como o ClientDataSet. Nesses casos, pode ser inviável a conversão de tais formatos em um padrão que pode ser lido por esses componentes. Além disso, se esses dados forem atualizados em curtos espaços de tempo, serão necessárias rotinas frequentes para atualização da sua exibição no aplicativo. Por esses motivos, é de suma importância conhecer os recursos para manipulação de componentes de listagem de dados por meio

de código, pois assim o desenvolvedor pode adaptar um formato específico de acordo com suas necessidades, não ficando dependente somente dos recursos visuais que o IDE fornece.

Baseado nisso, o objetivo do presente artigo é mostrar como construir um aplicativo no Delphi 10 Seattle utilizando o ListView, com todo o processamento e carga de dados sendo realizados via código, dispensando a utilização dos recursos gráficos para a vinculação dos dados, tal como LiveBindings. Explorando as funcionalidades desse componente, o exemplo proposto consiste em um aplicativo mestre-detalhe com dois ListViews: o primeiro irá listar as três principais divisões do campeonato brasileiro de futebol juntamente com os times pertencentes a cada divisão, apresentando o número de vitórias, derrotas e empates. Por outro lado, o segundo ListView terá o objetivo de apresentar os jogadores pertencentes a cada time, ou seja: quando um time é selecionado são listados os jogadores que o compõem. O exemplo desenvolvido também faz uso de orientação a objetos e dicionários para armazenar os dados, já que eles serão carregados para o ListView de forma dinâmica por meio de métodos.

Criando a interface gráfica

A primeira etapa para a construção deste exemplo é criar uma nova aplicação multidispositivo no Delphi 10 Seattle (*File > New > Multi-Device Application*) utilizando o layout em branco (*Blank Application*). No formulário automaticamente criado, que chamamos de *frmPrincipal*, deve ser adicionado um componente TTabControl (*tabPrincipal*) composto por duas guias (*tabTimes* e *tabJogadores*), como mostra a **Figura 1**.

O funcionamento do aplicativo seguirá a seguinte lógica: na primeira guia serão listadas três divisões do campeonato brasileiro (série A, série B e série C), bem como alguns dos times que pertencem a cada uma dessas divisões. Quando um determinado time for selecionado, a interface será direcionada para a segunda guia, na qual serão listados todos os jogadores que compõem o time correspondente, caracterizando um relacionamento mestre-detalhe. Além disso, na primeira guia será possível realizar pesquisas, enquanto que a segunda guia possibilitará que os jogadores sejam excluídos ou selecionados.

Para obter o mesmo design apresentado na **Figura 1**, na primeira guia (*tabTimes*) deve ser adicionada uma TToolBar (*tlbTimes*) com a propriedade *Align* configurada para *Top*. Dentro dessa toolbar deve ser adicionado um TButton (*btnPesquisa*) alinhado à direita, com a propriedade *StyleLookup* configurada com o valor *searchtoolbutton* para que uma imagem de lupa de pesquisa seja adicionada automaticamente ao botão.

O texto "Divisões e times" é adicionado por meio de um componente TLabel (*lblTitulo*) com a propriedade *Align* configurada para *Contents*, para que assim o rótulo ocupe toda a área da barra de ferramentas. Adicionalmente, esse componente tem a sua propriedade *Text* alterada para "Divisões e times", *TextSettings > HorzAlign* setado para *Center* e *TextSettings > Font >Style > fsBold* marcado como *True*. Essas três opções configuram qual texto será exibido, o alinhamento na horizontal e a ativação do negrito, respectivamente.

Por fim, um TListView (*lvwTimes*) deve ser adicionado com a propriedade *Align* marcada para *Client* para que o componente ocupe toda a área livre da guia. Sua propriedade *ItemAppearance > ItemAppearance* deve ter seu valor mudado para *ListItemRightDetail*, pois essa opção permitirá que um texto de detalhe seja adicionado à direita do texto principal.

A segunda guia (*tabJogadores*) é bastante semelhante. Alinhada ao topo encontra-se uma TToolBar (*tlbJogadoresCabecalho*), que contém um TButton (*btnVoltar*) alinhado à esquerda com a propriedade *StyleLookup* setada para *backtoolbutton*. Semelhante ao botão de pesquisa da primeira guia, essa configuração adiciona automaticamente a esse botão uma imagem, dessa vez com a seta de voltar. Um TLabel também é necessário (*lblJogadores*) com as mesmas configurações apresentadas anteriormente, porém com a diferença de que a propriedade *Text* deve possuir o valor "Jogadores <Time>". O texto entre os sinais de menor e maior será passado como parâmetro em tempo de execução, tendo a função de apresentar o nome do time correspondente que foi selecionado na guia anterior, como "Jogadores Coritiba", "Jogadores São Paulo", entre outros.

Outra ToolBar (*tlbJogadoresRodape*), mas com alinhamento inferior, também está presente, a qual contém dois TButtons (*btnExcluir* e *btnSelecionar*) com alinhamento à esquerda (*Align > Left*) e à direita (*Align > Right*), respectivamente. A propriedade *StyleLookup* para ambos deve ser configurada como *toolbutton*, e suas propriedades *Text* devem ser configuradas conforme a **Figura 1**. Um novo TListView (*lvwJogadores*) deve ser adicionado a essa guia, com a propriedade *Align* configurada para *Client* e a *ItemAppearance > ItemAppearance* mudada para *ListItemRightDetail*. Por último, um componente TActionList (*actAcoes*) também deve ser acrescentado, o qual será responsável por gerenciar a mudança das guias do TabControl.

É possível ainda observar à direita da **Figura 1** a janela *Structure*, que apresenta em detalhes a hierarquia dos componentes dessa interface gráfica. O uso dessa janela em aplicativos móveis é particularmente útil, pois, como geralmente existem vários componentes dentro uns dos outros, a visualização se torna mais fácil.

Da maneira que o TabControl está configurado, caso o aplicativo seja executado, as duas guias estariam visíveis para o usuário. Porém, neste exemplo o objetivo é que somente a primeira guia seja exibida e a segunda seja mostrada somente quando um registro for selecionado. Ou seja, as duas guias não poderão ser exibidas ao mesmo tempo, e para isso a propriedade *TabPosition* do TabControl (*tabPrincipal*) deve ser alterada para *None*. Por meio dessa alteração, somente uma das guias poderá ser acessada, e em tempo de desenvolvimento a navegação entre elas pode ser realizada por meio dos botões em forma de círculos localizados na parte inferior da janela.

Com um duplo clique no ActionList (*actAcoes*) abriremos o editor de ações conforme mostrado na **Figura 2**. Pode-se notar que foram definidas duas ações padrões do tipo *TChangeTabAction*, as quais têm a função de gerenciar as mudanças de guias automaticamente sem a necessidade de codificação para ocultá-las ou exibi-las.

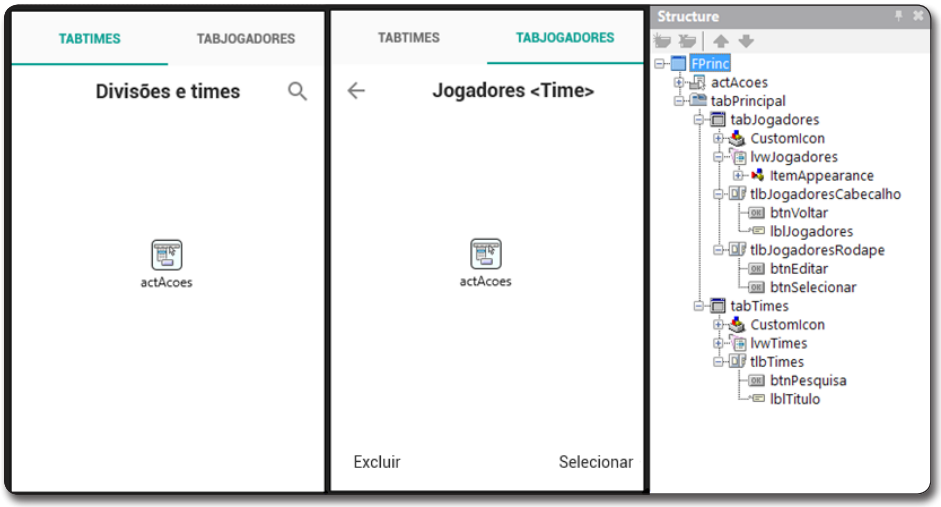

Figura 1. Interface gráfica do aplicativo

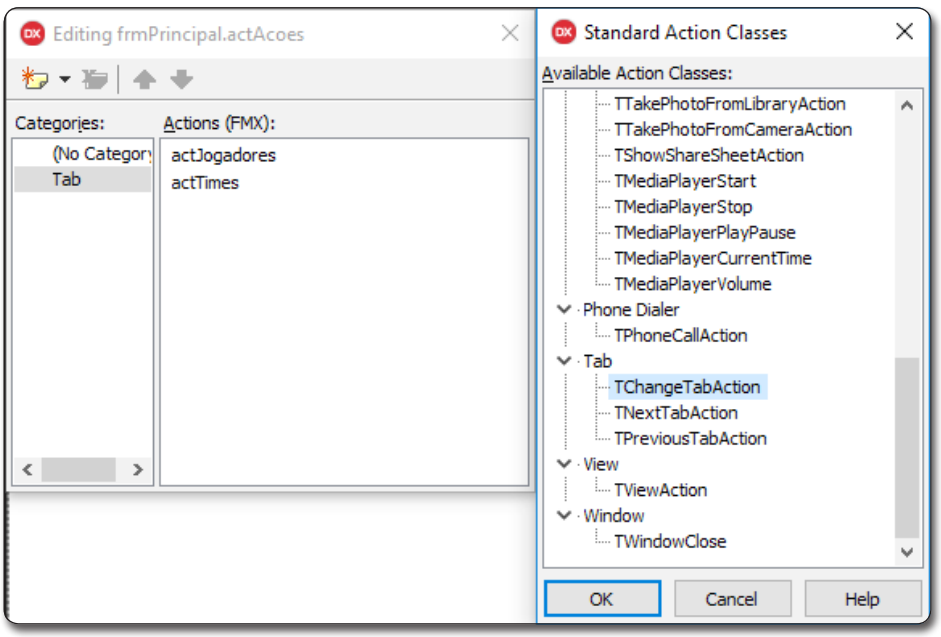

Figura 2. Ações para gerenciamento das guias

A única propriedade que precisa ser configurada para as ações é a propriedade *Tab*, a qual deve ter o valor *tabJogadores* para a ação *actJogadores* e *tabTimes* para a ação *actTimes*. Essa propriedade indica qual guia será aberta quando a ação for executada.

Populando o ListView principal com os times de futebol

Depois de finalizada a interface gráfica, o próximo passo é criar uma nova unit (*CBFTeamData)* por meio do comando *File > New > Unit - Delphi*, na qual serão definidas as classes, métodos e objetos que serão utilizados para a listagem dos dados no ListView. A **Listagem 1** apresenta a definição da classe TTime, que será responsável pela geração dos objetos relacionados aos times das séries A, B e C do campeonato brasileiro de futebol.

Entre as linhas 5 e 13 a classe é declarada, a qual possui quatro atributos, que dizem respeito ao nome do time, o número de vitórias, o número de derrotas e o número de empates. Todos esses atributos foram definidos com a visibilidade pública, ou seja, eles podem ser acessados diretamente pelos objetos sem a necessidade de propriedades ou métodos do tipo get/set para acesso aos seus valores. Na linha 12 é definida a assinatura do construtor da classe, que recebe como parâmetro quatro valores, um para cada atributo.

O código da **Listagem 2** deve ser incluído dentro da seção *interface* da unit, e nele são definidos os tipos, as constantes e as variáveis que serão utilizadas para a definição dos times e suas respectivas divisões.

Na linha 2 da **Listagem 2**, é definido um tipo para representar as três divisões, enquanto que nas linhas 6 e 7 é declarado um array constante de strings que armazenará os nomes das divisões, os quais serão exibidos no ListView. Na linha 3 é declarado um novo tipo, chamado *TClubes*, que tem a função de armazenar todos os times que farão parte deste exemplo, ou seja, cinco times para cada uma das três divisões.

Entre as linhas 8 e 13 é definida uma nova constante (*CBFConstante*), que é representada por uma matriz de dimensões 3x5 (conforme ilustra a **Figura 3**), ou seja, três linhas para representar as divisões e cinco colunas para representar os times de cada divisão. É importante salientar que o número 3, que representa as linhas da matriz, é referente ao tamanho do tipo *TDivisao* (linha 2). Dessa forma, é possível definir quais são os times que pertencem à série A (linha 10), à série B (linha 11) e à série C (linha 12).

Para ilustrar e facilitar o entendimento dos valores declarados para a constante *CBFConstante* (linha 8), a **Figura 3** apresenta uma matriz contendo as divisões do campeonato, representadas pelo índice *i*, e seus respectivos times, representados pelo índice *j*. Por meio dessa figura é possível observar a estrutura utilizada para armazenar os times, da qual os dados serão extraídos para posterior preenchimento do ListView.

Nota

A unit Generics.Collections, declarada na **Listagem 1**, possui implementações de classes para manipulação de matrizes (arrays), listas, pilhas, filas, entre outros. Por sua vez, a unit System.SysUtils possui várias classes, tipos e funções utilitárias para o manuseio de strings, conversão e comparação de dados, arquivos e diretórios.

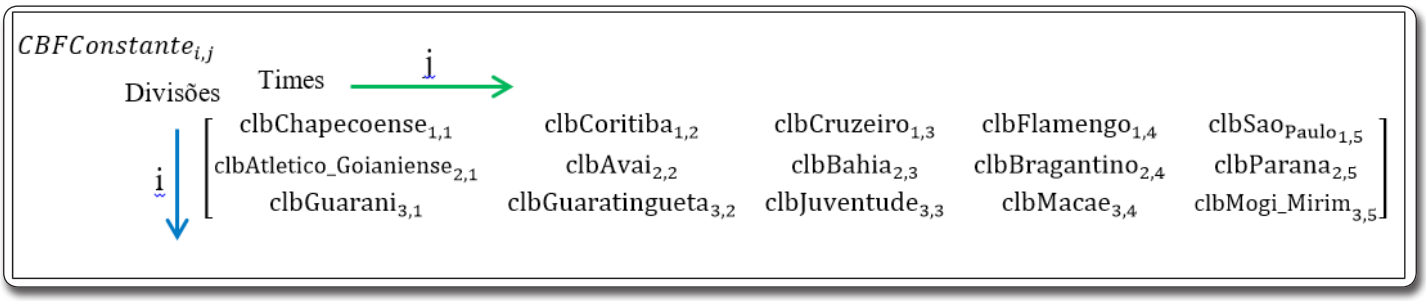

Figura 3. Dados da CBFConstante

Listagem 1. Definição da classe TTime

```
01 unit CBFTeamData;
02 interface
03 uses
04  Generics.Collections, System.SysUtils;
05 type
06  TTime = class
07  public
08   Nome: string;
09   Vitorias: Integer;
10  Derrotas: Integer;
11  Empates: Integer;
12  constructor Create(const Nome: string; Vitorias, Derrotas, Empates: Integer);
13 end;
14 implementation
15 constructor TTime.Create(const Nome: string; Vitorias, Derrotas,
   Empates: Integer);
16 begin
17  Self.Nome := Nome;
18  Self.Vitorias := Vitorias;
19  Self.Derrotas := Derrotas;
20  Self.Empates := Empates;
21 end;
```

Listagem 2. Definição dos tipos de dados e constantes

```
01 type
02  TDivisao = (divPrimeira, divSegunda, divTerceira);
03  TClubes = (clbChapecoense, clbCoritiba, clbCruzeiro, clbFlamengo,
    clbSao_Paulo,
04          clbParana, clbAtletico_Goianiense, clbAvai, clbBahia, clbBragantino,
05          clbGuarani, clbGuaratingueta, clbJuventude, clbMacae, clbMogi_Mirim);
06 const
07  DivisaoNomes: array[TDivisao] of string = ('Série A', 'Série B', 'Série C');
08  CBFConstante: array[TDivisao, 1..5] of TClubes =
09  (
10   (clbChapecoense, clbCoritiba, clbCruzeiro, clbFlamengo, clbSao_Paulo),
11   (clbAtletico_Goianiense, clbAvai, clbBahia, clbBragantino, clbParana),
12   (clbGuarani, clbGuaratingueta, clbJuventude, clbMacae, clbMogi_Mirim)
13  );
14 var
15  TimeInfo: TDictionary<TClubes, TTime>;
```

Por fim, na linha 15 da **Listagem 2** é declarada uma variável do tipo TDictionary (**BOX 1**) chamada *TimeInfo*, que possibilita representar um conjunto de chaves e valores, onde o tipo TClubes representa a chave e o tipo TTime representa o valor. Com isso, a chave será utilizada para realizar as buscas enquanto o valor é o campo que será retornado para a aplicação.

BOX 1. TDictionary

A classe TDictionary faz parte do namespace System.Generics.Collections e tem o objetivo de representar estruturas que contenham chaves e valores, bastante semelhantes a uma matriz. Nesse tipo de estrutura o tempo de resposta é menor, já que ela possui otimizações específicas para buscas, e os dados estão organizados de forma que pesquisas são feitas pela chave e a resposta ou retorno é dado pelo valor.

A **Listagem 3** apresenta o procedimento com o código que efetivamente cria os objetos e/ou times. Na linha 3 é declarado um objeto do tipo *TTime*, que é utilizado para a criação de todos os quinze times utilizados no exemplo. Na linha 5 é chamado o construtor da classe, que recebe como parâmetro o nome do time, o número de vitórias, o número de derrotas e o número de empates.

Listagem 3. Criação dos times/objetos

```
01 procedure PopularTimeInfo;
02 var
03  Time: TTime;
04 begin
05  Time := TTime.Create('Chapecoense', 12, 15, 11);
06  TimeInfo.Add(clbChapecoense, Time);
07  Time := TTime.Create('Coritiba', 11, 16, 11);
08  TimeInfo.Add(clbCoritiba, Time);
09  Time := TTime.Create('Cruzeiro', 15, 13, 10);
10  TimeInfo.Add(clbCruzeiro, Time);
11  Time := TTime.Create('Flamengo', 15, 19, 4);
12  TimeInfo.Add(clbFlamengo, Time);
13  // Parte do código dessa listagem foi omitido.
14 end;
```

Em continuação do procedimento, na linha 6 a constante correspondente ao clube e o objeto recém-criado contendo as propriedades do time são atribuídos ao dicionário *TimeInfo*, composto pela chave/valor *TClubes* e *TTime*, respectivamente. Dessa maneira, ao acessar o dicionário por meio das constantes dos clubes é possível obter as propriedades definidas para cada time. Esse mesmo procedimento é realizado mais quatorze vezes, para todos os times da aplicação.

O código a seguir apresenta o que será executado na inicialização da unit, isto é, a seção *initialization* (linha 1), que é executada no momento que a aplicação é iniciada, permitindo adicionar instruções que serão executadas previamente:

```
01 initialization
02  TimeInfo := TDictionary<TClubes, TTime>.Create;
03  PopularTimeInfo;
```

Na linha 2 a variável *TimeInfo* (definida na linha 15 da **Listagem 2**) é efetivamente instanciada, e, na linha 3, o procedimento *PopularTimeInfo* (**Listagem 3**) é invocado para preencher todos os dados dos times. É importante citar que essa seção deve ser codificada na parte final da unit.

De volta ao formulário com a interface gráfica, é necessário definir o procedimento *CarregaTimes* para efetivamente popular o ListView (*lvwTimes*) com os dados do dicionário. A **Listagem 4** mostra a declaração desse procedimento na seção *private* do formulário. Apertando as teclas *CTRL+SHIFT+C* a seção com a implementação do procedimento é criada, conforme mostrado na **Listagem 5**. Entre as linhas 3 e 7 são declaradas as variáveis locais que serão utilizadas para popular o ListView, que, com exceção da variável *I* (linha 3) e da variável *Item* (linha 5), tiveram seus tipos explanados na unit *CBFTeamData*.

Nota

A classe TListViewItem implementa os itens do ListView, os quais podem ter particularidades na representação gráfica. Isso ocorre pelo fato de possuírem uma ou mais características de apresentação, como uma legenda, um texto detalhe, uma imagem, um ícone ou um botão. Caso o RAD Studio não consiga localizar essa classe, o namespace FMX.ListView.Appearances deve ser adicionado ao uses.

Na linha 9 é dado início ao processo para popular os dados para o ListView por meio do método *BeginUpdate*, que tem a função de evitar o redesenho do ListView na tela até que a operação de inclusão seja finalizada. Após o término da adição de itens, o método *EndUpdate* (linha 30) é chamado para realizar a atualização do componente em tela. É também importante salientar que com a utilização do método *BeginUpdate* antes das mudanças e do *EndUpdate* depois delas é possível obter grande performance.

Na linha 11 é codificada uma estrutura de repetição para percorrer todas as divisões cadastradas, ou seja, esse comando *for* será executado três vezes, e, dentro dele, os times correspondentes a cada série do campeonato brasileiro serão adicionados. Na linha 13 um novo item é adicionado ao ListView (*lvwTimes*), e posteriormente, na linha 14, o array *DivisaoNomes* é utilizado para obter o nome da divisão de acordo com a variável *D*, que contém a divisão do campeonato, sendo que seu valor é alterado a cada interação do loop. Por exemplo, quando a variável *D* contiver o valor *divPrimeira* será retornado o nome da divisão, ou seja: Série A.

Na linha 15 é definido o propósito desse item dentro do ListView, que é ser um cabeçalho para agrupar os times. Devido ao fato de que para este exemplo foram definidos cinco times para cada divisão do campeonato brasileiro, a estrutura de repetição presente na linha 16 contém cinco iterações. O objetivo dela é adicionar cada time à sua divisão correspondente, e na linha 18 a variável *Clube* recebe o valor correspondente ao índice da matriz *CBFConstante*, sempre variando o valor de *I* para buscar um time diferente, conforme demostrando pela matriz apresentada na **Figura 3**. Dessa forma, nas primeiras execuções, por exemplo, serão retornados todos os times da primeira divisão, pois eles estão armazenados na primeira linha da matriz. Na linha 19, por meio do método *TryGetValue* é obtido do dicionário *TimeInfo* as propriedades do time: nome, vitórias, derrotas e empates, por meio dos parâmetros *Clube* e *Time*.

Listagem 4. Declaração de CarregaTimes no formulário

```
01 private
02  procedure CarregaTimes;
03 public
04 end;
```

Listagem 5. Carregamento dos times para o ListView

```
01 procedure TfrmPrincipal.CarregaTimes;
02 var
03  I: Integer;
04  D: TDivisao;
05  Item: TListViewItem;
06 Clube: TClubes;
07 Time: TTime;
08 begin
09 lvwTimes.BeginUpdate;
10 try
11   for D := Low(TDivisao) to High(TDivisao) do
12   begin
13    Item := lvwTimes.Items.Add;
14    Item.Text := DivisaoNomes[D];
15    Item.Purpose := TListItemPurpose.Header;
16    for I := 1 to 5 do
17    begin
18     Clube := CBFConstante[D, I];
19     if TimeInfo.TryGetValue(Clube, Time) then
20     begin
21      Item := lvwTimes.Items.Add;
22      Item.Text := Time.Nome;
23      Item.Detail := IntToStr(Time.Vitorias) + 'V - ' + IntToStr(Time.Derrotas)
        + 'D - ' + IntToStr(Time.Empates) + 'E';
24      Item.Accessory := TAccessoryType.More;
25      Item.Tag := Ord(Clube);
26     end;
27    end;
28   end;
29 finally
30  lvwTimes.EndUpdate;
31 end;
32 end;
```

Semelhante ao código para adicionar o cabeçalho, na linha 21 um novo item é adicionado ao ListView, na linha 22 o nome do time é setado e na linha 23 é feita uma concatenação para mostrar o número de vitórias, derrotas e empates como informação de detalhe. Na linha 24 é definida a imagem de uma seta como acessório para indicar que mais informações podem ser visualizadas. Por fim, o código da linha 25 atribui ao item um número para a propriedade *Tag* por meio da função *Ord*, que tem o objetivo de retornar um valor inteiro para um tipo ordinal, neste caso o *Clube* passado como parâmetro. O objetivo dessa linha é atribuir um identificador único a cada item, para que posteriormente a listagem dos jogadores de cada time seja realizada de maneira eficaz.

Um último detalhe que ainda precisa ser feito para poder realizar o primeiro teste efetivo do aplicativo é programar o evento *OnCreate* do formulário, conforme mostrado na **Listagem 6**. Na linha 3 é setada a guia inicial e/ou ativa, enquanto que na linha 4 é chamado o procedimento para carregar os times para o ListView.

A **Figura 4** apresenta o resultado da execução do aplicativo em um dispositivo Android, na qual pode-se observar o nome das séries e logo abaixo cada um dos times definidos anteriormente, juntamente com a totalização dos números de vitórias, derrotas e empates. À direita da **Figura 4**, pode-se notar que o campo de pesquisa está visível, e para que essa função seja habilitada deve-se programar o evento *OnClick* do botão correspondente (*btnPesquisa*) conforme demonstrado na **Listagem 7**. Na linha 3 é utilizado o operador de negação *not* para habilitar ou desabilitar a função de pesquisa do ListView (*lvwTimes*), de acordo com o clique no botão. Com isso, para pesquisar algum registro o usuário primeiramente deve clicar nessa opção para que tal funcionalidade seja habilitada, enquanto que um novo clique no botão desabilita essa funcionalidade.

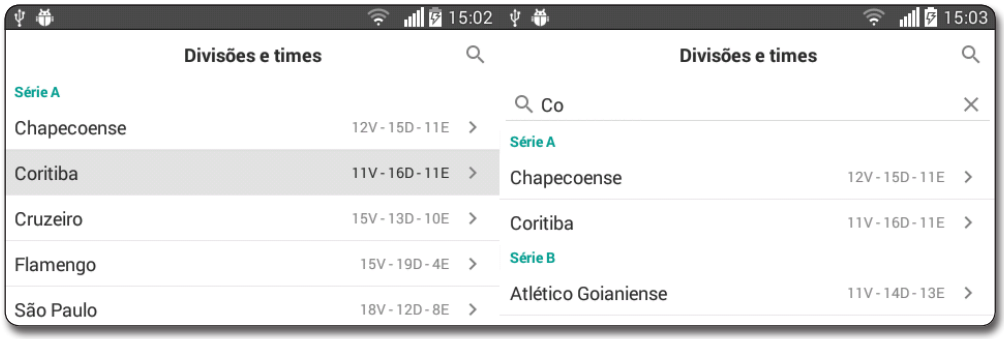

Figura 4. Execução da guia times

Listagem 6. OnCreate do formulário

```
01 procedure TfrmPrincipal.FormCreate(Sender: TObject);
02 begin
03   tabPrincipal.ActiveTab := tabTimes;
04   CarregaTimes;
05 end;
```

Listagem 7. Habilitação do botão de pesquisa

```
01 procedure TfrmPrincipal.btnPesquisaClick(Sender: TObject);
02 begin
03   lvwTimes.SearchVisible := not lvwTimes.SearchVisible;
04 end;
```

Nota

Consulte a seção **Links** para informações sobre como fazer o deploy do aplicativo para um dispositivo Android.

Populando o ListView de detalhe com os jogadores

A seção anterior apresentou os passos para popular o ListView da primeira guia com os dados dos times. Esta seção, por sua vez, tem o objetivo de mostrar os passos necessários para que os jogadores de cada um dos times sejam apresentados no ListView da segunda guia (*lvwJogadores*). Para isso, na seção *interface* da unit criada no tópico anterior (*CBFTeamData*) deve ser definida a classe para armazenar os dados dos jogadores, conforme é apresentado na **Listagem 8**.

Listagem 8. Criação da classe TJogador

```
01 TJogador = class
02 public
03   Numero: Integer;
04   Nome: string;
05   constructor Create(Numero: Integer; const Nome: string);
06 end;
07 constructor TJogador.Create(Numero: Integer; const Nome: string);
08 begin
09   Self.Numero := Numero;
10  Self.Nome := Nome;
11 end;
```

De forma similar à classe TTime, a classe TJogador possui dois atributos públicos (linhas 3 e 4), um para indicar o número da camisa e outro para indicar o nome do jogador. Está presente também um construtor, que recebe como parâmetro ambos os campos (linha 5). Entre as linhas 7 e 11, encontra-se a implementação desse construtor, que assim como na classe TTime somente faz a atribuição dos parâmetros recebidos para o objeto. É importante frisar que a classe TJogador deve ser definida na seção *interface* da unit, enquanto que o código do construtor deve estar localizado na seção *implementation*.

Uma nova variável deve ser adicionada na seção de declaração de variáveis da unit, conforme pode ser visto no código a seguir, na linha 3:

```
01 var
02   TimeInfo: TDictionary<TClubes, TTime>;
03   TimeJogadores: TDictionary<TClubes, TObjectList<TJogador>>;
```

Essa variável é também um dicionário, porém ela armazena além dos clubes, no primeiro parâmetro, uma lista de jogadores por meio da classe TObjectList (**BOX 2**). Em resumo, essa variável tem a função de armazenar os times vinculados com a lista de seus jogadores correspondentes, simulando o mestre-detalhe.

BOX 2. TObjectList

A classe TObjectList faz parte do namespace System.Generics.Collections e tem a função de armazenar qualquer tipo de objeto em memória, sendo que isso é possibilitado graças ao fato de que qualquer classe faz herança com a classe base TObject. Ela é comumente utilizada para o gerenciamento e armazenamento de listas de objetos.

Após ter definido a variável que armazenará os jogadores, é necessário neste momento preencher esse dicionário com os jogadores correspondentes de cada time. Entre as linhas 1 e 8 da **Listagem 9** foi definida uma função para gerar os nomes dos jogadores automaticamente, para que o nome de cada um não precise ser definido manualmente. Essa função recebe como parâmetro o tipo *TClubes* e retorna uma lista de objetos do tipo TJogador. Na linha 5 é criada a lista de objetos, e na linha 6 é definida uma estrutura de repetição com 22 interações, pois neste exemplo estão sendo considerados 22 jogadores para cada time. Por fim, na linha 7 um novo jogador é criado e adicionado ao retorno da função. O formato do nome do jogador segue a seguinte estrutura: "Jogador 1 – Coritiba", "Jogador 2 – São Paulo" e assim sucessivamente.

Listagem 9. Populando o dicionário de jogadores

```
01 function TimeJogador(Clube: TClubes): TObjectList<TJogador>;
02 var
03   I: Integer;
04 begin
05   Result := TObjectList<TJogador>.Create;
06   for I := 1 to 22 do
07     Result.Add(TJogador.Create(I, 'Jogador ' + IntToStr(I)
       + ' - ' + TimeInfo.Items[Clube].Nome));
08 end;
09  procedure PopularTimeLista;
10 var
11  Jogador: TObjectList<TJogador>;
12  Clube: TClubes;
13 begin
14  for Clube := Low(TClubes) to High(TClubes) do
15    begin
16      Jogador := TimeJogador(Clube);
17      TimeJogadores.Add(Clube, Jogador);
18  end;
19 end;
```

Ainda nessa listagem, entre as linhas 9 e 19 foi criado o procedimento *PopularTimeLista*, que tem a função de adicionar os jogadores de cada clube no dicionário *TimeJogadores* (definido anteriormente). A estrutura de repetição da linha 14 percorre todos os quinze times, e na linha 16 a variável *Jogador* (que é uma lista de jogadores) recebe o retorno da função *TimeJogador*. Com isso, essa variável conterá todos os jogadores do time passado como parâmetro, para que na linha 17 a lista de jogadores seja adicionada ao clube correspondente.

Por fim, o último passo na unit *CBFTeamData* é inicializar a execução desses procedimentos na seção *initialization* da unit, conforme apresentado em negrito na **Listagem 10** (linhas 3 e 5). Na linha 3 o dicionário é instanciado em memória, enquanto que na linha 5 ele é populado por meio dos métodos anteriormente apresentados.

Voltando ao formulário, é preciso definir um procedimento para carregar os jogadores para o ListView (*lvwJogadores*). Para isso, na seção *private* do formulário deve ser criado o procedimento *CarregaJogadores*, conforme mostrado na linha 3 da **Listagem 11**. Basicamente, esse procedimento recebe como parâmetro uma variável do tipo *TClubes*, faz a busca pelos jogadores do clube e popula o ListView com o nome dos jogadores. Inicialmente, na linha 12 é necessário limpar o ListView para que os dados de outro time não sejam visíveis, e na linha 15 é feita uma pesquisa para verificar se os valores passados por parâmetro para o método *TryGetValue* existem no dicionário, similarmente ao que foi executado para popular os times. A estrutura de repetição da linha 17 percorre todos os jogadores que estão na lista e os adicionam ao ListView, em um processo bastante semelhante ao executado anteriormente. Neste caso, o nome do jogador aparecerá no texto do ListView enquanto que o número será mostrado no detalhe (linhas 20 e 21).

Listagem 10. Inicialização da unit

```
01 initialization
02  TimeInfo := TDictionary<TClubes, TTime>.Create;
03  TimeJogadores := TDictionary<TClubes, TObjectList<TJogador>>.Create;
04  PopularTimeInfo;
05  PopularTimeLista;
```

Listagem 11. Carregamento dos jogadores para o ListView

```
01 private
02   procedure CarregaTimes;
03   procedure CarregaJogadores(Clube: TClubes);
04 public
05 end;
06 procedure TfrmPrincipal.CarregaJogadores(Clube: TClubes);
07 var
08   I: Integer;
09   Item: TListViewItem;
10  Jogador: TObjectList<TJogador>;
11 begin
12  lvwJogadores.Items.Clear;
13  lvwJogadores.BeginUpdate;
14  try
15    if TimeJogadores.TryGetValue(Clube, Jogador) then
16    begin
17      for I := 0 to Jogador.Count - 1 do
18      begin
19        Item := lvwJogadores.Items.Add;
20        Item.Text := Jogador[I].Nome;
21        Item.Detail := 'Número: ' + IntToStr(Jogador[I].Numero);
22      end;
23    end;
24  finally
25    lvwJogadores.EndUpdate;
26  end;
27 end;
```

A última etapa para que os jogadores sejam exibidos quando um time for selecionado é programar o evento *ItemClick* do ListView *lvwTimes*, pois conforme mencionado anteriormente quando um time for selecionado os jogadores correspondentes devem ser carregados dinamicamente no ListView *lvwJogadores*. A **Listagem 12** mostra esse procedimento. Na linha 5 inicialmente é chamada a ação para que o controle da aplicação seja direcionado para a segunda guia, e na linha 6 é recuperado o clube selecionado no ListView *lvwTimes* por meio da propriedade *Tag*. Na linha 7 o texto do TLabel é preenchido com a string "Jogadores" + o nome do time correspondente, utilizando os

sinais de maior e menor (< >) anteriormente definidos no *lblJogadores*. Por exemplo, caso seja selecionado o time São Paulo, aparecerá no componente "Jogadores São Paulo". Por fim, na linha 8 é invocado o procedimento *CarregaJogadores*, passando como parâmetro o time selecionado pelo usuário.

Listagem 12. Evento ItemClick do ListView

```
01 procedure TfrmPrincipal.lvwTimesItemClick(const Sender: TObject;
     const AItem: TListViewItem);
02 var
03   Clube: TClubes;
04 begin
05   actJogadores.ExecuteTarget(Self);
06   Clube := TClubes(AItem.Tag);
07   lblJogadores.Text := 'Jogadores ' + TimeInfo.Items[Clube].Nome;
08   CarregaJogadores(Clube);
09 end;
```

Com a finalização desse código o aplicativo já está com suas funcionalidades básicas prontas. A **Figura 5** apresenta a janela do software rodando em um dispositivo Android, na qual podemos observar a listagem com o nome dos jogadores no texto do ListView e o número como informação de detalhe. Um ponto importante a frisar é que por padrão o ListView adiciona um botão acessório com uma seta para a visualização de mais detalhes. Para alterar o tipo de botão é necessário acessar a propriedade *Item Appearance>ItemObjects>Acessory>AcessoryType*. Porém, essa opção permite somente a troca da imagem do botão, não existindo uma opção *None* na qual o botão não existiria. Para que a imagem seja ocultada, deve-se acessar a propriedade *ItemAppearance>Item Objects>Acessory>Visible* e mudar seu valor para *False*. Essa opção deve ser alterada também para a propriedade *ItemAppearance> ItemEditObjects>Acessory>Visible*, para que a imagem seja também ocultada quando o registro está em edição.

Botões de voltar, excluir e selecionar

Os itens restantes para a finalização do presente exemplo são as três últimas configurações de interface, ou seja, os botões voltar, excluir e selecionar. Atualmente, caso um time seja selecionado para a exibição de seus jogadores, não é possível voltar para a guia principal em tempo de execução, e para que essa funcionalidade esteja disponível basta que a propriedade *Action* do botão *btnVoltar* seja configurada para a ação *actTimes*. Como as ações definidas anteriormente foram criadas com base na ação padrão *TChangeTabAction*, não é necessário codificar a mudança de guias.

A **Listagem 13** apresenta o código para selecionar os registros, que deve ser programado no evento *OnClick* do botão *btnSelecionar*. Na linha 3 é alterada a aparência do ListView para quando o mesmo se encontrar em modo de edição, e neste código é definida a aparência *ListItemRightDetailShowCheck* para que um checkbox seja adicionado ao lado de cada registro. Na linha 4 o modo de edição do ListView é ativado ou desativado de acordo com o operador *not* e baseado em quando o usuário clicar no botão, ou seja, um clique ativa o modo de edição, enquanto que outro clique em sequência o desativa. É importante salientar que o código da **Listagem 13** somente habilita a seleção de registros, desse modo caso alguma operação necessite ser executada sobre os registros selecionados uma codificação adicional seria necessária.

Nota

Consulte a seção **Links** para mais informações sobre as configurações de aparência do ListView.

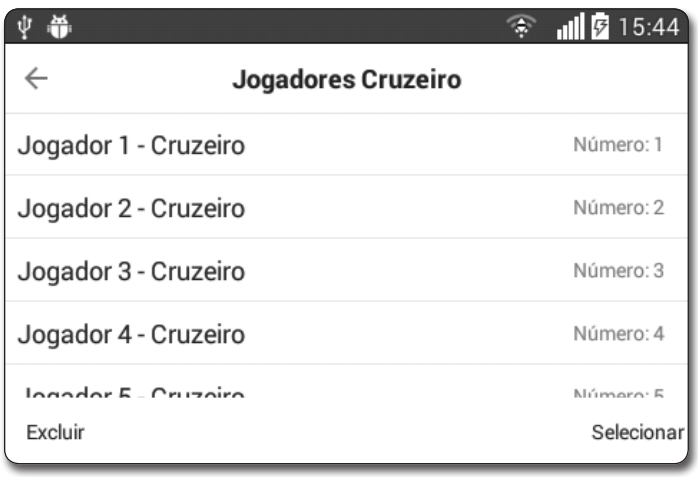

Figura 5. Execução da guia jogadores

Listagem 13. Seleção de registros

```
01 procedure TfrmPrincipal.btnSelecionarClick(Sender: TObject);
02 begin
03   lvwJogadores.ItemAppearance.ItemEditAppearance :=
       'ListItemRightDetailShowCheck';
04   lvwJogadores.EditMode := not lvwJogadores.EditMode;
05 end;
```

A última etapa é codificar o botão de exclusão (*btnExcluir*) no evento *OnClick*, conforme mostrado na **Listagem 14.** Esse código inicialmente verifica se existe algum item selecionado (linha 3) e na linha 6 é chamado o método para excluir o registro, passando como parâmetro o índice do item. Caso nenhum registro esteja selecionado uma mensagem de aviso é mostrada para o usuário (linhas 8 e 9).

Listagem 14. Exclusão de registros

```
01 procedure TfrmPrincipal.btnExcluirClick(Sender: TObject);
02 begin
03 if Assigned(lvwJogadores.Selected) then
04  begin
05   with lvwJogadores do
06    Items.Delete(Items.IndexOf(Selected));
07  end
08  else
09   ShowMessage('Nenhum jogador selecionado');
10 end;
```

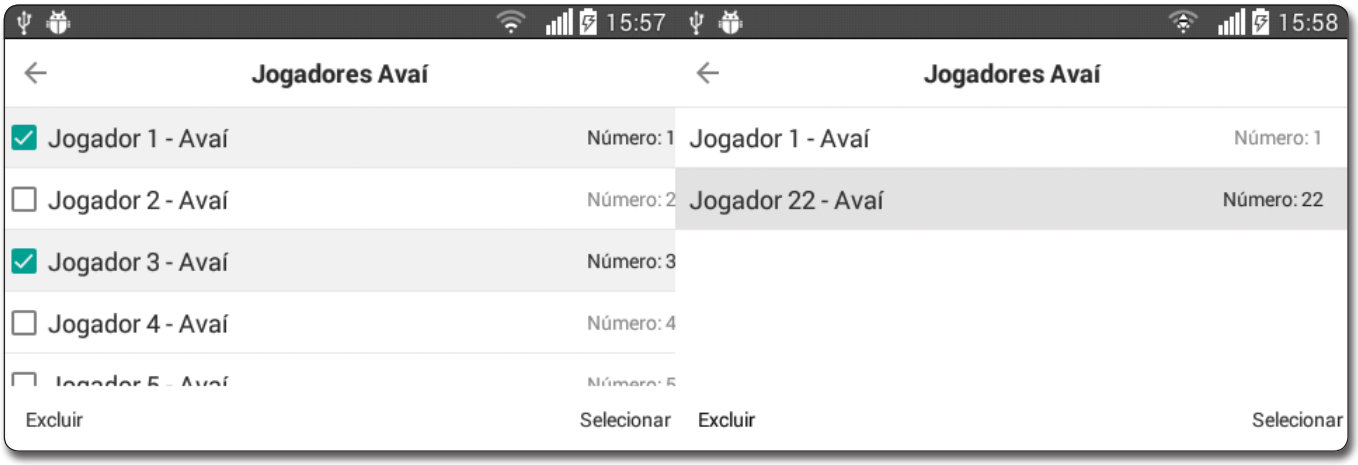

Figura 6. Seleção e exclusão de registros

A **Figura 6** apresenta a execução do aplicativo, mostrando as opções de seleção à esquerda e exclusão à direita. Pode-se observar que os jogadores 1 e 3 foram selecionados, enquanto que, com exceção dos jogadores 1 e 22, todos os outros foram excluídos na direita da figura.

O conhecimento dessas técnicas, sem a utilização dos recursos visuais do tipo RAD, é de fundamental importância, principalmente em situações nas quais não existe uma fonte de dados adequada para a manipulação das informações por meio dos recursos gráficos. Dessa forma, caso os dados estejam armazenados nas nuvens ou em algum outro tipo de formato, é necessário carregar e manipular os dados para um ListView utilizando linhas de código. Como pôde ser observado, o ListView é um componente bastante poderoso que possibilita diversas formas de personalização, cabendo ao desenvolvedor utilizar as melhores estratégias e configurações para cada situação específica.

Autor

Jones Granatyr

Doutorando em informática e mestre em ciência da computação, ambos na área de inteligência artificial. Trabalha em projetos de pesquisa relacionados à área de inteligência artificial, tais como sistemas especialistas, mineração de dados, mineração de textos e sistemas multiagente.

Autor

Aldo Marcelo Paim

Bacharel em sistemas de informação, especialista em banco de dados, administração e desenvolvimento. É mestrando em informática pela Pontifícia Universidade Católica do Paraná (PUCPR), atuando na área de mineração de dados e aprendizagem de máquina. Tem experiência de 10 anos com a linguagem Delphi.

Links e Referências:

Aparências do ListView
http://docwiki.embarcadero.com/Libraries/Seattle/en/FMX.ListView.TAppearanceListView.ItemEditAppearanceName

Documentação da Embarcadero
http://docwiki.embarcadero.com

Realizando deploy para um aplicativo Android
http://www.embarcadero.com/starthere/seattle/mobdevsetup/android/en/running_your_android_application_on_an_android_device.html

Effectively Using List Controls in Object Pascal Mobile Apps
http://delphi-insider.blogspot.com.br/2015/03/effectively-using-list-controls-in.html

W. Duarte. "Delphi para Android e iOS: Desenvolvendo Aplicativos Móveis". Rio de Janeiro: Brasport, 2015.

J. Granatyr, G. Pauli, T. L. Lauxen, J. P. Barddal, F. Spak, F. Taffe. "Construindo Aplicativos Android no Delphi 10 Seattle", Clube Delphi, v. 166, 2016.

Técnicas de debug no Delphi

Ferramentas de depuração para melhorar sua produtividade

Durante o desenvolvimento de uma aplicação, é comum a ocorrência de erros de variados tipos e origens. Quando se trata de um erro de sintaxe ou que fere alguma regra estrutural da linguagem de programação ou das ferramentas utilizadas, normalmente o próprio editor/IDE aponta a localização do problema, de forma que o programador pode facilmente corrigi-lo. Por outro lado, erros de lógica ou de regras de negócio normalmente só são localizados durante o processo de utilização ou de testes, já com a aplicação em execução, portanto sua identificação e correção é mais difícil.

Quando os próprios desenvolvedores ou testadores não identificam esses problemas, o software acaba sendo entregue ao cliente com bugs, que provavelmente só serão descobertos durante a utilização prática. Em um cenário ainda pior a aplicação pode não apresentar erros que interrompam seu fluxo de funcionamento, mas gerar internamente resultados que não correspondam aos esperados. Esse tipo de problema normalmente só é encontrado posteriormente e, dependendo das estratégias utilizadas para o armazenamento e backup dos dados, bem como do volume de informações geradas, pode ser impossível de ser corrigido. Os prejuízos dessas falhas podem ser os mais variados, tanto para o cliente quanto para o responsável pelo desenvolvimento do sistema, devido à atribuição de uma imagem negativa ao seu produto/serviço.

Identificar um bug pode ser uma tarefa árdua para principiantes ou desenvolvedores sem conhecimentos de boas técnicas de debug, por isso é importante conhecer as ferramentas que o IDE disponibiliza para auxiliar nessa tarefa.

Uma das técnicas mais utilizadas na depuração de uma aplicação é a exibição de mensagens em pontos estratégicos (com a função *ShowMessage*, por exemplo). O problema dessa prática é que é muito fácil se esquecer de remover essas mensagens após o *debug*, fazendo com que sejam exibidas para o usuário final e tornando necessária uma nova correção e liberação da aplicação.

Uma forma de otimizar essa técnica, evitando essas mensagens indesejadas para o usuário final, é utilizar a diretiva *{$IFDEF DEBUG}*, que faz com que todo código em seu interior seja executado apenas para os programas gerados em modo *debug*, e não nas versões *release*. Pode-se utilizar essa instrução não apenas para mensagens, mas também para criar um log específico ou executar comandos necessários para algum teste, como a criação de um arquivo de configuração. Porém, apesar de muito útil, essa diretiva não resolve todos os nossos problemas. Realizar todos os processos de depuração baseados apenas nela será trabalhoso e nada produtivo.

Para que possamos realizar o debug de forma produtiva e eficiente, vamos conhecer neste artigo algumas ferramentas disponíveis no Delphi fundamentais para todos os desenvolvedores, desde os mais iniciantes até os mais avançados.

Fique por Dentro

Este artigo irá mostrar a importância do debug na vida do desenvolvedor e como utilizar as ferramentas de depuração disponíveis no Delphi. Conhecimentos de boas técnicas de debug podem auxiliar a localizar de forma mais fácil e eficiente a causa de um problema, seja ele de lógica ou de regra de negócio. Esse tipo de técnica garante ao desenvolvedor maior produtividade, por acelerar a localização dos erros, e ao software maior qualidade, por auxiliar na redução dos problemas.

Breakpoints

Breakpoints são pontos de interrupção utilizados para pausar a execução de uma aplicação e permitir que se acompanhe, linha a linha, o fluxo do código a partir dali. Um breakpoint funciona apenas quando a aplicação está sendo executada em modo de depuração e pode ser criado de várias formas:

• Clicando na barra do lado esquerdo do editor de código, ao lado da linha em que se deseja adicionar o breakpoint;
• Clicando com o botão direito do mouse na linha onde o breakpoint deve ser adicionado e, no menu suspenso, clicando em *Debug > Toggle Breakpoint*;
• Pressionando F5 com o cursor estando posicionado na linha em que se deseja adicionar o breakpoint.

A **Figura 1** ilustra a adição de um breakpoint. Repare que do lado esquerdo do editor um ponto vermelho indica a linha onde ocorrerá a parada.

Nota

Essas orientações são baseadas no funcionamento padrão do Delphi. A instalação de plug-ins/extensões pode alterar esses caminhos/atalhos.

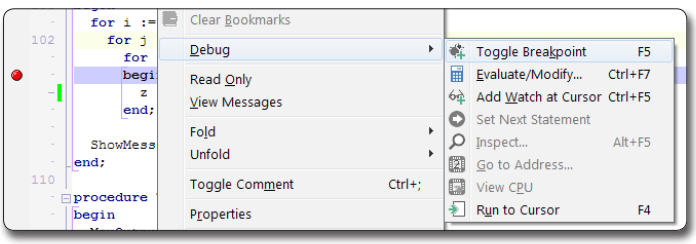

Figura 1. Adicionando um breakpoint

Com o breakpoint definido, ao executar a aplicação o fluxo do código será interrompido ao atingir a linha marcada. Nesse momento, podemos acompanhar, passo a passo, as próximas instruções. Para isso, basta utilizar os comandos *Step Over* (F8), para executar a próxima linha, *Trace Into* (F7), para entrar no método em execução na atual linha, ou *Run* (F9), para executar todo o código até o próximo breakpoint.

Para que possamos realizar um pequeno teste, crie uma nova aplicação no Delphi, adicione um *TButton* e um *TLabel* e, em seguida, adicione o comando da **Listagem 1** no evento *OnClick* do botão.

Nesse código criamos uma variável do tipo string e, na linha 4, atribuímos a ela o valor "Teste de breakpoint". Na linha 5 atribuímos o valor dessa variável ao *caption* do label.

Adicione então um breakpoint na linha 4 e execute a aplicação (clicando no menu *Run > Run*, equivalente ao atalho F9). Ao clicar no botão, a aplicação será pausada e o editor do Delphi será aberto com a linha onde o breakpoint foi adicionado marcada em destaque. Pressione F8 e observe que a seta do lado esquerdo avançará para a próxima instrução, informando o próximo comando a ser executado. Cada vez que esse comando for utilizado, a linha selecionada será executada e o fluxo avançará para a próxima.

Ao parar em um breakpoint, além da possibilidade de ver o algoritmo sendo executado passo a passo, pode-se ver os valores das variáveis e as propriedades de um objeto apenas passando o mouse sobre eles, como mostra a **Figura 2**. Nesse caso, o cursor do mouse foi parado sobre a variável *sValue*.

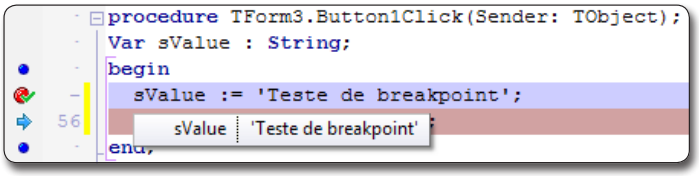

Figura 2. Verificando o valor de uma variável

Listagem 1. Evento OnClick do botão para testar o breakpoint

```
01 procedure TForm1.Button1Click(Sender: TObject);
02 Var sValue : String;
03 begin
04   sValue := 'Teste de breakpoint';
05   lblMsg.Caption := sValue;
06 end;
```

Os breakpoints podem ser representados por vários ícones diferentes, dependendo do seu estado atual. Na **Figura 3** vemos uma ilustração desses ícones e logo a seguir o status que cada um representa.

1. Válido e habilitado, porém com o debugger inativo;
2. Válido, habilitado e com debugger ativo, indicando que ocorrerá uma parada neste ponto;
3. Inválido e habilitado, indicando que não haverá parada neste ponto, normalmente por ser uma linha em branco ou um comentário;
4. Válido, desabilitado e com debugger inativo;
5. Válido, desabilitado e com debugger ativo;
6. Inválido e desabilitado.

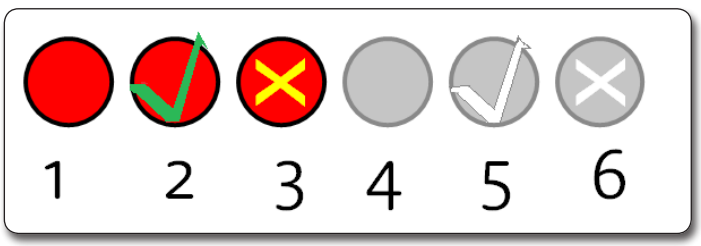

Figura 3. Possíveis ícones de um breakpoint

Um breakpoint também pode ser configurado para funcionar apenas se uma determinada condição for atendida. Para acessar essa configuração, devemos clicar com o botão direito do mouse sobre o ícone do breakpoint e, em seguida, na opção *Breakpoint Properties*. Será aberta uma janela como a da **Figura 4**, onde devemos alterar a propriedade *Condition*.

Uma situação comum para uso do campo *Condition* é na depuração de uma rotina que envolva a iteração sobre uma coleção de dados, como linhas de uma tabela ou de um arquivo. Nesses casos, se observarmos que um erro ocorre durante o processamento dessa lista de registros, pode ser inviável acompanhar passo a passo o debug até chegar na linha problemática. Para evitar isso, podemos estabelecer alguma condição de parada do breakpoint, como o número do registro ou o valor de um campo.

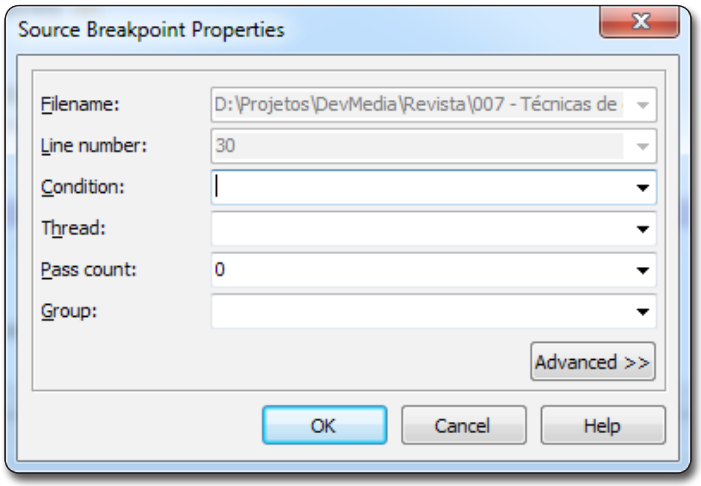

Figura 4. Breakpoint Properties

Nessa tela também está presente o campo *Thread*, que permite depurar uma thread específica, devendo ser preenchido com o seu id quando for o caso. Já o campo *Pass count* indica o número de vezes que a linha onde está o breakpoint deve ser processada até que ocorra a parada. Por fim, o campo *Group* atua de forma a agrupar vários pontos de parada com base em uma característica em comum a ser definida pelo programador. Esse agrupamento permitirá, por exemplo, que todos os breakpoints de um grupo sejam habilitados ou desabilitados de uma só vez na Breakpoint List.

Breakpoint List

A Breakpoint List pode ser acessada a partir das teclas de atalho Ctrl + Alt + B ou através do menu *View* > *Debug Windows* > *Breakpoints*, e nela encontramos todos os breakpoints existentes na aplicação, que podem ser removidos, editados ou desabilitados. Como pode ser visto na **Figura 5**, essa lista exibe informações como o arquivo ou endereço do breakpoint, a linha, a condição de parada e a thread onde ocorrerá a depuração. Ao dar um duplo clique sobre um dos itens seremos direcionados para o trecho de código onde o breakpoint se encontra. Para alterar um item, podemos pressionar *Enter* sobre ele ou clicar com o botão direito do mouse e, em seguida, na opção *Breakpoint Properties*.

Para habilitar ou desabilitar um grupo de breakpoints inteiro, clique com o botão direito do mouse sobre a lista e, no menu *Breakpoint*, utilize as opções *Enable Group* ou *Disable Group*.

Evaluate/Modify

A janela Evaluate/Modify permite que se avalie e modifique o valor de uma variável ou expressão durante a depuração do código. Para acessá-la, basta utilizar o atalho *CTRL + F7* quando a aplicação estiver em modo de debug.

Como podemos ver na **Figura 6**, ao inserir uma expressão qualquer no campo *Expression* (que pode ser o nome de uma variável, por exemplo), seu valor é apresentado no campo *Result*. Para testar esse recurso, você pode adicionar um breakpoint na linha 5 da **Listagem 1** e executar a aplicação. Quando a execução for pausada, selecione no editor a variável *sValue* e pressione *CTRL + F7*. Automaticamente a variável será avaliada e seu valor, exibido. Caso deseje alterá-lo durante o debug, preencha o campo *New value*. Esse último recurso é bastante útil para testar hipóteses sem que a aplicação tenha que ser fechada e reaberta.

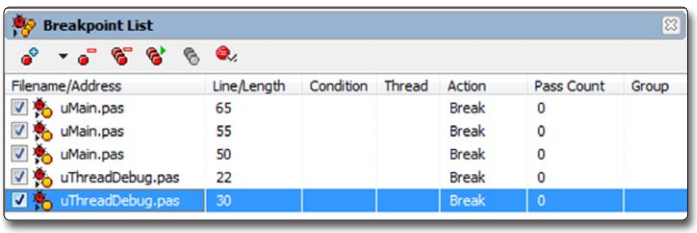

Figura 5. Janela de Breakpoints

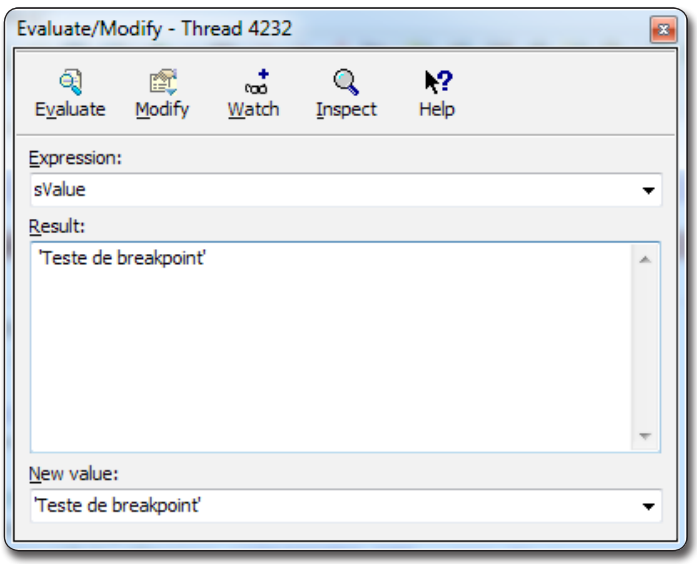

Figura 6. Janela Evaluate/Modify

Como dito anteriormente, no campo *Expression* também podem ser inseridas expressões, desde as mais simples até as instruções mais complexas. Para testar, adicione a instrução *ShowMessage(sValue)* a esse campo e verá uma mensagem sendo exibida na tela. Essa possibilidade é especialmente útil quando se está trabalhando com *StringLists* ou documentos em memória, pois se pode utilizar o método *SaveToFile* para salvar seu conteúdo em arquivo e assim conferir o resultado em disco.

Watch List

A Watch List pode ser acessada a partir das teclas de atalho *CTRL + ALT + W* ou através do menu *View* > *Debug Windows* > *Watches*. Ela permite criar uma lista de elementos a serem monitorados em tempo real, acompanhando a mudança dos seus valores.

Para adicionar um elemento à Watch List pode ser utilizado o botão *Watch* da janela de Evaluate/Modify, visto no topo da **Figura 6**, ou o atalho *CTRL + F5* com algum elemento selecionado no editor de código do Delphi.

A **Figura 7** mostra alguns valores sendo avaliados: o primeiro é uma variável local e o segundo é o *caption* de um label.

Se esse atalho for acionado sem que haja algum código selecionado, será aberta a janela *Watch Properties* para definir o watch, como pode ser visto na **Figura 8**. Nessa tela, além de informar uma expressão a ser monitorada, podemos criar um *group* e definir o tipo do elemento monitorado. Observe que, ao criar um watch com um grupo, uma nova aba na janela de *Watch List* será criada, permitindo que todos os watches de um grupo sejam habilitados ou desabilitados de uma só vez como nos breakpoints.

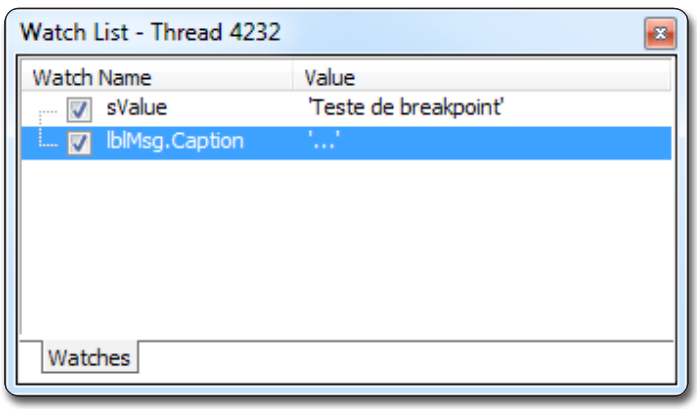

Figura 7. Janela de Watch List

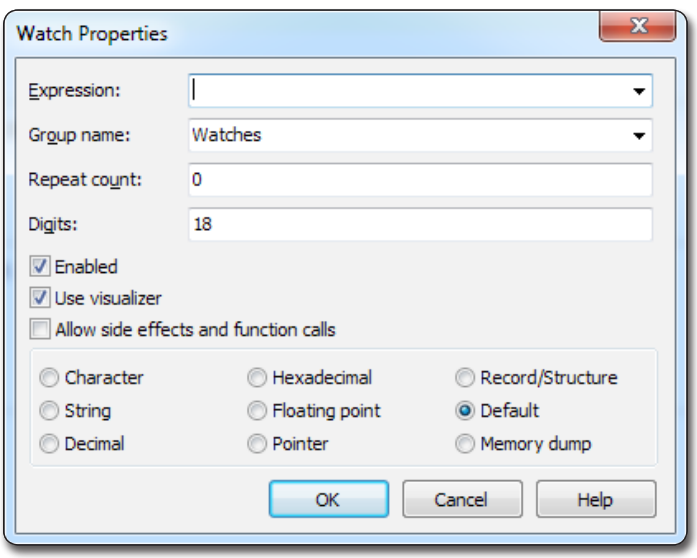

Figura 8. Janela Watch Properties

Um recurso muito útil ao se monitorar loops, principalmente loops encadeados, é o *Repeat count*. Caso estejamos lidando com um loop de dois níveis, ao monitorar o incrementador do segundo nível podemos colocar o valor de *Repeat count* como 2. Isso fará com que na *Watch List* seja apresentado um array na coluna de value, onde o primeiro elemento representa o valor do incrementador que está sendo monitorado e o segundo elemento é referente ao incrementador do primeiro nível do loop.

A janela *Watch Properties* também pode ser acessada através da *Watch List*, clicando com o botão direito em um watch e, em seguida, na opção *Edit Watch*.

Local Variables

Ao falar sobre watches não se pode deixar de falar sobre o recurso Local Variables. Local Variables é uma Watch List automática de todos os elementos locais à um objeto, que pode ser acessada através das teclas de atalho *Ctrl + Alt + L* ou através do menu *View > Debug Windows > Local Variables*. Ao abrir a tela de Local Variables, são apresentadas as variáveis do método em execução, porém também temos acesso às variáveis do objeto utilizado, como pode ser visto na **Figura 9**.

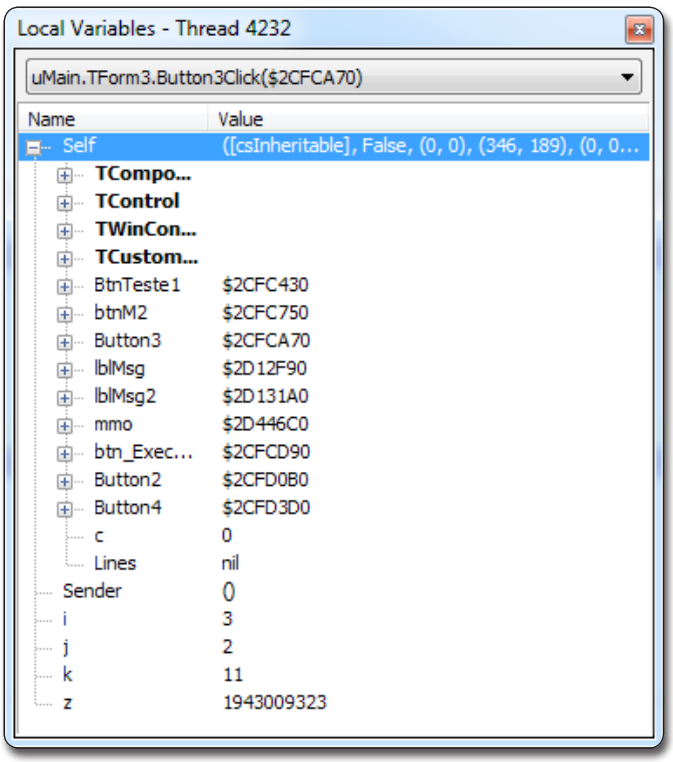

Figura 9. Janela de Local Variables

Esse recurso evita a criação de watches desnecessários de elementos simples em um método ou objeto, porém não descarta o uso de watches, pois diversos recursos, como adicionar expressão, definir um laço ou criar grupos, não estão disponíveis para Local Variables.

Event Log

O Event Log pode ser acessado a partir das teclas de atalho *Ctrl + Alt + V* ou através do menu *View > Debug Windows > Event Log*. O Event Log lista informações, como o início de threads, exceções, paradas em breakpoints e mensagens do debugger, e é útil para vários truques de depuração. Um desses truques é um breakpoint disparar uma mensagem no Event Log, o que é útil para evitar disparadas de métodos *ShowMessage* e criar logs dos acontecimentos.

Para criar um log através do debugger basta, na janela *Breakpoint Properties*, clicar no botão *Advanced*. A janela será expandida e novas opções serão apresentadas. O campo *Log message* pode ser utilizado para disparar mensagens ao Event Log para que saibamos quando a execução passou por esse ponto, enquanto que o campo *Eval expression* serve para fazer uma chamada a um método ou expressão. Abaixo dele existe um checkbox chamado *Log result*, que, quando marcado, faz com que o resultado da expressão seja também exibido no Event Log.

Uma outra forma de criar um log no Event Log é com uso do método *OutputDebugString(<MinhaMensagem>)*, a partir do qual podemos disparar um log em qualquer ponto do nosso sistema.

Por fim, o Event Log, através de sua janela de configuração, pode ser configurado para exibir apenas mensagens desejadas, limpar o log ou até mesmo ter suas cores de fundo e linhas alteradas. Essa janela, apresentada na **Figura 10**, pode ser acessada através do menu *Tools* > *Options* > *Debugger Options* > *Event Log*.

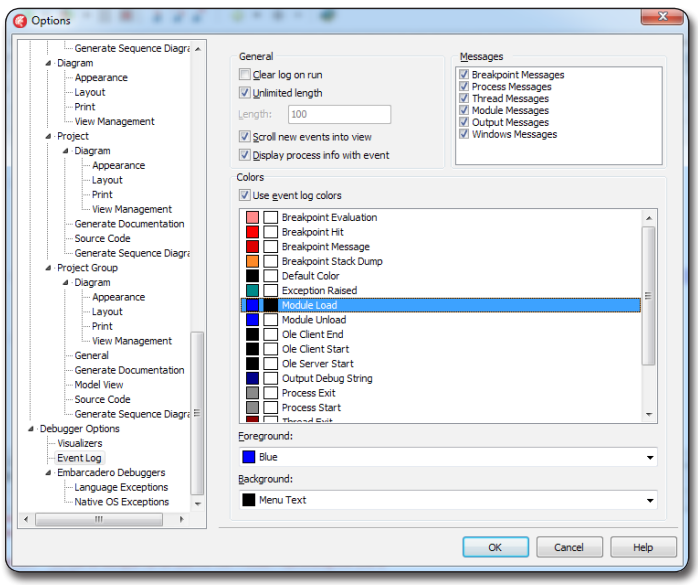

Figura 10. Janela de configuração do Event Log

Debug Inspector

O Debug Inspector, apresentado na **Figura 11**, pode ser acessado a partir das teclas de atalho *Alt + F5* ou através do botão *Inspect*, presente na janela Evaluate/Modify. Ele é um Object Inspector de tempo de execução, isto é, mostra as propriedades e métodos de um objeto, muito útil quando se deseja visualizar todo o conteúdo de um objeto em tempo de execução.

CPU

A janela de CPU pode ser acessada através das teclas de atalho *Ctrl + Alt + C* ou através do menu *View* > *Debug Windows* > *CPU Windows* > *Entire CPU*. Através dessa janela é possível acessar o conteúdo da memória, dos registradores do processador, e o disassembly do código que está sendo executado no momento.

Apesar de oferecer uma visão muito baixo nível e ser pouco utilizada, pode-se observar (veja a **Figura 12**) que é fácil identificar um bloco de código Delphi e, por consequência, o seu assembly, que está logo abaixo.

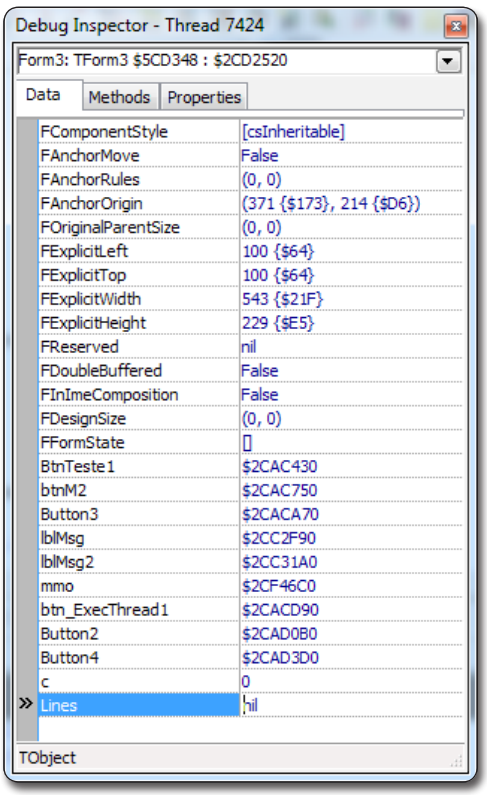

Figura 11. Janela do Debug Inspector

```
005B9547 894DEC          mov [ebp-$14],ecx
005B954A 8955E8          mov [ebp-$18],edx
005B954D 8945FC          mov [ebp-$04],eax
005B9550 33C0            xor eax,eax
005B9552 55              push ebp
005B9553 683D965B00      push $005b963d
005B9558 64FF30          push dword ptr fs:[eax]
005B955B 648920          mov fs:[eax],esp
uMain.pas.78: for i := 0 to 10 do
005B955E 33C0            xor eax,eax
005B9560 8945F8          mov [ebp-$08],eax
uMain.pas.79: for j := 0 to 10 do
005B9563 33C0            xor eax,eax
005B9565 8945F4          mov [ebp-$0c],eax
uMain.pas.80: try
005B9568 33C0            xor eax,eax
005B956A 55              push ebp
005B956B 68A8955B00      push $005b95a8
005B9570 64FF30          push dword ptr fs:[eax]
005B9573 648920          mov fs:[eax],esp
uMain.pas.81: OutputDebugString ('PASSANDO AKI -->>> ') ;
005B9576 684C965B00      push $005b964c
005B957B E89C7FE5FF      call OutputDebugString
uMain.pas.83: ShowMessage('Teste');
005B9580 B880965B00      mov eax,$005b9680
005B9585 E80A1FF9FF      call ShowMessage
uMain.pas.84: OutputDebugString ('PASSANDO AKI -->>> ') ;
005B958A 684C965B00      push $005b964c
005B958F E8887FE5FF      call OutputDebugString
uMain.pas.87: k := i div j;
005B9594 8B45F8          mov eax,[ebp-$08]
005B9597 99              cdq
```

Figura 12. Janela de CPU

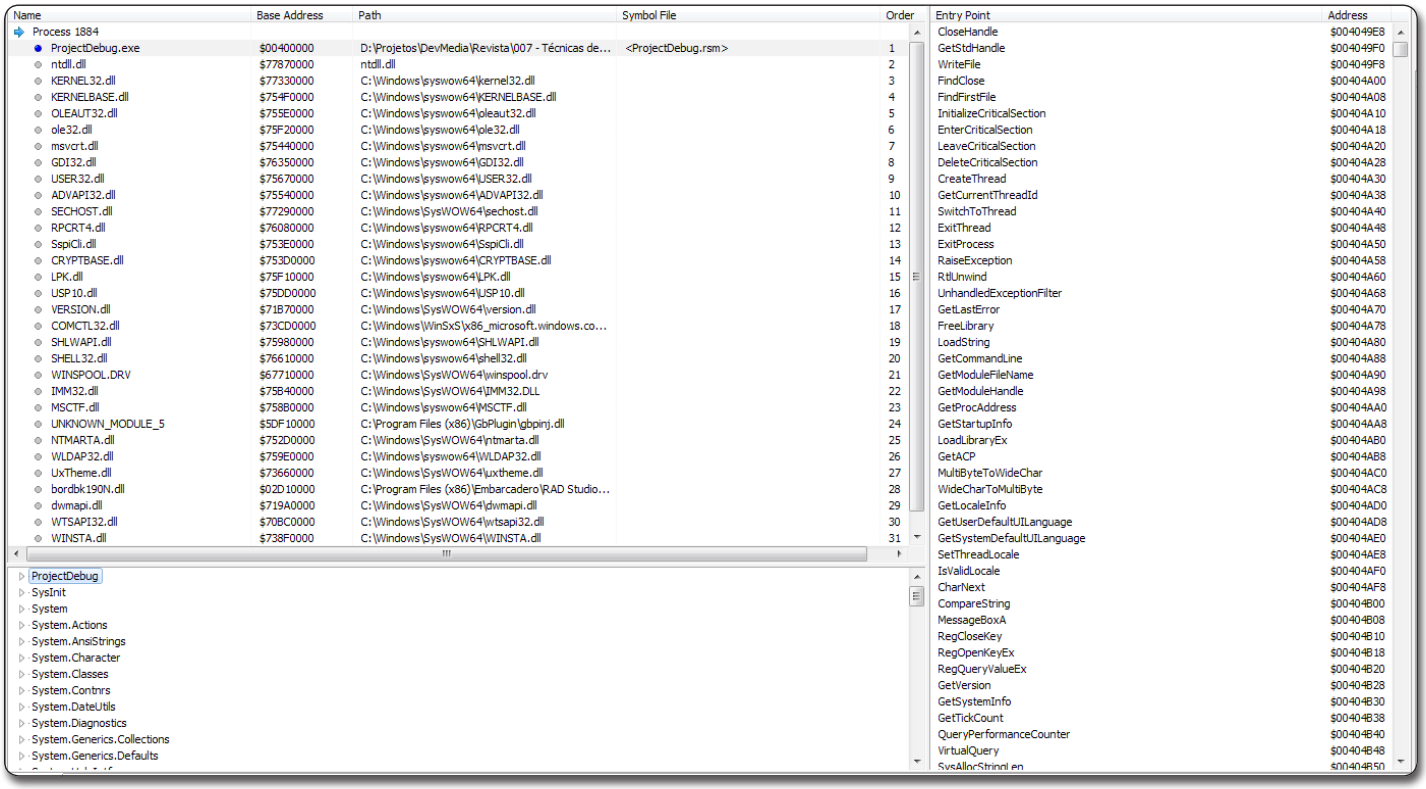

Figura 13. Janela Modules

Além disso, tem recursos muito interessantes, que podem ser acessados clicando o botão auxiliar do mouse sobre o endereço desejado. Alguns desses recursos são: o salto de endereços, por meio do atalho *Ctrl + G*, e o New EIP (*Extended Instruction Pointer*), que permite efetuar saltos para blocos desejados, possibilitando que, ao entrar em uma instrução não desejada, possa ser feito um salto para uma nova instrução sem a necessidade de fechar e reabrir o programa. A janela de CPU pode ser vista na **Figura 12**.

Modules

A janela Modules, vista na **Figura 13**, contém informações importantes para aplicações que trabalham modularizadas ou carregam bibliotecas externas através de BPL (*Borland Package Library*) ou DLL (*Dynamic-link library*), e pode ser acessada através das teclas de atalho *Ctrl + Alt + M* ou do menu *View > Debug Windows > Modules*. Ela apresenta, além do módulo carregado, o caminho completo do módulo e sua lista de funções, e permite definir um break a um módulo ou adicionar um novo módulo.

Call Stack

Call Stack é um recurso que exibe a pilha de chamadas das rotinas que foram executadas até o ponto que a aplicação se encontra (veja a **Figura 14**). Apesar de não muito intuitivo, o Call Stack é muito útil para compreender qual foi o caminho percorrido para se chegar ao atual ponto. Ele pode ser acessado através das teclas de atalho *Ctrl + Alt + S* ou através do menu *View > Debug Windows > Call Stack*.

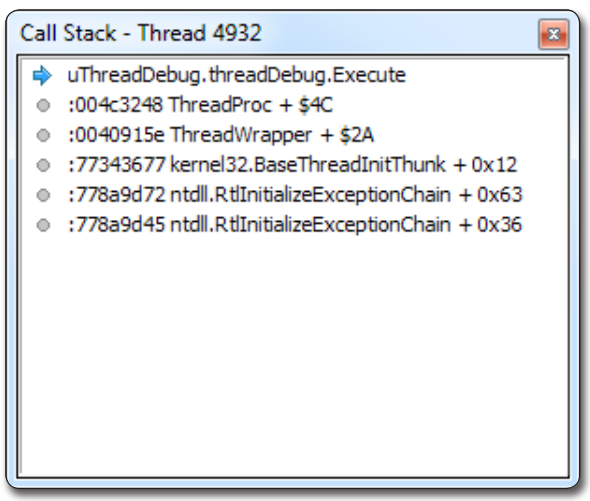

Figura 14. Janela Call Stack

Thread Status

Para debugar com mais eficiência sistemas com threads, está disponível o Thread Status, recurso que pode ser acessado através das teclas de atalho *Ctrl + Alt + T* ou do menu *View > Debug Windows > Threads*. Na janela de Thread Status é possível acompanhar todas as threads do processo e seus status, além de ser possível saltar entre as threads e pausar uma ou mais threads. Ao clicar com o botão auxiliar do mouse sobre um item será aberto um menu de contexto onde serão apresentadas algumas opções, como a *Make Current*, que define a thread como corrente.

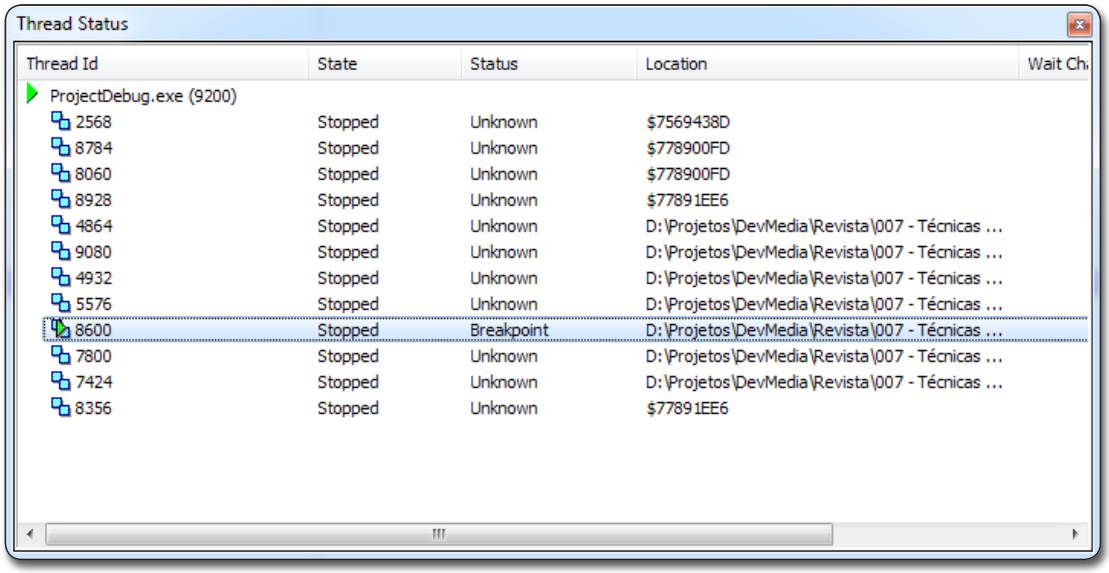

Figura 15. Janela Thread Status

Para facilitar debugar apenas uma thread em um contexto onde existe um número elevado de threads rodando, pode-se parar as thread não desejadas com a opção *freeze* ou *freeze all other*. Para continuar executando uma thread congelada, utilize a opção *thaw* ou *thaw all*. Na **Figura 15** pode ser vista a janela Thread Status.

Recursos avançados do Breakpoint

Como foi observado, a *Breakpoint Properties* contém um botão *Advanced*, que permite a configuração de mais recursos. O primeiro campo da seção *Actions* (seção aberta com o clique no botão *Advanced*) é o checkbox *Break*, que habilita ou desabilita a parada quando a execução passa por um breakpoint. Esse tipo de recurso é muito útil para que possam ser feitas ações sem nenhuma interrupção na aplicação. Podemos utilizá-lo, por exemplo, juntamente com a opção *Log Message*, resultando na disparada de uma mensagem ao Log toda vez que esse Breakpoint for invocado, sem gerar a interrupção da aplicação. Em conjunto, pode-se definir as condições que esse breakpoint irá ser invocado, e, se desejar, a *Eval expression* para validar alguma expressão ou até mesmo executar algum método. Essas combinações de ações permitem uma enorme gama de possibilidades de uso.

Os Breakpoints ainda têm outros recursos, como o "Ignore subsequent exceptions" e "Handle subsequent exceptions". Ao utilizar o Ignore, as exceções que são levantadas em debug não serão mais levantadas, e apenas exceções de runtime serão processadas, ou seja, uma exceção devidamente tratada, que geralmente é levantada no debug, mas não apresenta erro de fato na execução, não será mais apresentada. O Handle é o contrário, isto é, todas as exceções serão levantadas, então, se deseja saber se existe alguma exceção tratada que pode estar ocasionando algum problema, essa opção é muito útil. Como as duas opções são conflitantes, elas não podem ser utilizadas juntas.

Utilizando os recursos de depuração

Limpando um arquivo de log somente no modo debug

Para colocar em prática algumas das ferramentas vistas até o momento, implemente no projeto criado no início deste artigo um método como o da **Listagem 2**.

Listagem 2. Adiciona mensagem ao arquivo de Log

```
01 procedure TForm1.AddLog(psMsg : String);
02 var
03   sFile : String;
04   oLog  : TStringList;
05 Begin
06   sFile := ExtractFilePath(Application.ExeName) + '\MeuLog.log';
07   oLog  := TStringList.Create;
08   try
09    oLog.LoadFromFile(sFile);
10    oLog.Add(psMsg);
11    oLog.SaveToFile(sFile);
12   finally
13    FreeAndNil(oLog);
14   end;
15 end;
```

Esse método será utilizado para adicionar texto a um arquivo chamando *MeuLog.log*, contido na pasta do exe (esse arquivo deverá ser criado manualmente visto que o algoritmo da **Listagem 2** não prevê a falta dele). Nas linhas 9 a 13 é carregado para o objeto *Log*, de *TStringList*, o conteúdo do *sFile*, e, em seguida, uma string é adicionada na última linha desse arquivo e o objeto é destruído após o arquivo ser salvo.

Depois, crie o método da **Listagem 3**, o qual será responsável por limpar o arquivo de log. Esse método é muito semelhante ao da **Listagem 2**, a diferença está na linha 10, na qual, ao invés de adicionar uma linha, uma string vazia é atribuída para todo o conteúdo do arquivo.

Listagem 3. Limpa o arquivo de Log

```
01 procedure TForm1.ClearLog;
02 var
03   sFile : String;
04   oLog  : TStringList;
05 Begin
06   sFile := ExtractFilePath(Application.ExeName) + '\MeuLog.log';
07   oLog  := TStringList.Create;
08   try
09    oLog.LoadFromFile(sFile);
10    oLog.Text := '';
11    oLog.SaveToFile(sFile);
12   finally
13    FreeAndNil(oLog);
14   end;
15 end;
```

Após a criação dos métodos, adicione dois botões ao formulário, onde um fará a chamada ao método de *ClearLog* e o outro, ao método de *AddLog*, passando como parâmetro "Teste...". Além disso, é necessária a criação de uma variável de escopo global do tipo boolean, com o nome "DebugMode", que será usada no método *create* do formulário conforme a **Listagem 4**. Essa listagem fará com que no *create* do formulário a variável global *DebugMode* seja atribuída como verdadeira caso a execução esteja em modo de debug,

Na linha 6 do código da **Listagem 4** adicione um Breakpoint e deixe as configurações como definido na **Tabela 1.**

Listagem 4. Método create do formulário

```
01 procedure TForm1.FormCreate(Sender: TObject);
02 begin
03   {$IFDEF DEBUG}
04    DebugMode := True;
05   {$ENDIF}
06 end;
```

Propriedade	Valor
Condition	DebugMode
Group	Debug
Break	False
Log message	DebugMode Ativo
Eval expression	ClearLog()
Log result	False

Tabela 1. Definição do breakpoint

Com a variável *DebugMode* como verdadeira, será disparado o Log message e executado o método *ClearLog* sem gerar interrupções na execução, já que o checkbox break está desativado. Digite alguma informação no arquivo de log e salve-o, execute a aplicação e, com a tela de Event Log aberta, note que foi criado um evento de Breakpoint message e outro de Breakpoint expression.

Além disso, como esperado, o conteúdo do arquivo de log foi apagado.

Nesse exemplo temos uma aplicação que limpa o arquivo de log somente se estiver em modo debug e, sendo isso uma configuração exclusiva da máquina do desenvolvedor, não existe perigo de se esquecer de remover o código e ocasionar problemas no cliente.

Adicionando conteúdo a um arquivo de log

Para dar continuidade ao exemplo, adicione um novo botão e um label ao formulário. Ao evento de clique do botão adicione o conteúdo da **Listagem 5**.

No método da **Listagem 5** é carregado o arquivo *Lista.txt*, que também deve estar previamente criado. Depois, um loop percorre as linhas desse arquivo, acumulando o valor de suas linhas na variável *nValue*. Ao final, o valor da soma das linhas é atribuído ao label *lblMsg2*. Esse tipo de operação causa muitos erros, pois os valores presentes nas linhas podem não ser um inteiro válido, por isso a necessidade de debugar. Na linha 14 adicione um Breakpoint e configure-o como definido na **Tabela 2**.

Dessa forma teremos um breakpoint que não irá parar a aplicação nem irá gerar um Event Log, mas irá fazer uso de recursos de expression para gerar conteúdo no arquivo externo de log apenas em modo de debug. Para testar, basta que se execute a aplicação e pressione o botão *BtnPercorrerItens*.

Listagem 5. Ler itens do arquivo

```
01 procedure TForm1.btnPercorrerItensClick(Sender: TObject);
02 Var
03   oLocalFile : TStringList;
04   i     : Integer;
05   nValue : Integer;
06   sLinha : String;
07 begin
08   oLocalFile := TStringList.Create;
09   try
10    oLocalFile.LoadFromFile(ExtractFilePath(Application.ExeName) + '\Lista.txt');
11    nValue := 0 ;
12    sLinha := '';
13    for i := 0 to oLocalFile.Count-1 do
14     nValue := nValue + oLocalFile[i].ToInteger();
15
16    lblMsg2.Caption := nValue.ToString();
17   finally
18    FreeAndNil(oLocalFile);
19   end;
20 end;
```

Propriedade	Valor
Condition	
Group	Log
Break	false
Log message	
Eval expression	AddLog('Linha:' + IntToStr(i) + ' = '+ oLocalFile[i]);
Log result	false

Tabela 2. Definição do breakpoint

Adicionando um evento ao Event Log

Para demonstrar como adicionar um evento ao Event Log, criaremos um loop que, em modo debug, irá exibir suas linhas no Event Log. Para isso, deve ser criado um método chamado *OpDS*, que recebe um parâmetro do tipo string (*psMsg*) e implementa a seguinte linha: *"OutputDebugString(pChar('Output :' + psMsg));"*. Esse método encapsulará as saídas realizadas para o Event Log, adicionando um evento ao mesmo. No método de limpar log adicione a linha *"OpDS('limpando log')*; " , que fará uso do método recém-criado.

Depois, crie um botão e uma label o nomeie-os como *btnLoop* e *lblLoop*, respectivamente, e implemente o evento de clique do botão conforme a **Listagem 6**.

Na linha 5 adicione um Breakpoint e o configure-o conforme a **Tabela 3**.

Ao executar a aplicação e clicar sobre o botão *btnLoop*, perceba que o salto dos eventos ocorre de 10 em 10 como definido pelo *Pass count*. Altere o valor de *Pass count* para 0 e faça um novo teste: como resultado não existe mais um salto no contador i.

Nota

Para facilitar a leitura do Event Log, no menu Tools > Options > Event Log, desmarque os checkboxes de Messages, deixando marcado apenas a opção Output Messages. Isso fará com que apenas mensagens de output sejam exibidas na lista de Event Log.

Listagem 6. Loop

```
01 procedure TForm1.btnLoopClick(Sender: TObject);
02 var i : integer;
03 begin
04   for i := 0 to 1000 do
05     lblLoop.Caption := i.ToString();
06 end;
```

Propriedade	Valor
Pass count	10
Group	Loop
Break	False
Log message	
Eval expression	OpDS('Item -> ' + IntToStr(i))
Log result	False

Tabela 3. Definição do breakpoint do loop

Habilitando/desabilitando grupos de breakpoints

O breakpoint conta, ainda, com um outro recurso muito interessante, que é a opção de desabilitar ou habilitar grupos de breakpoint através de um breakpoint, por exemplo: se o breakpoint "A" foi acionado, ele automaticamente desabilita o Grupo X; se o Breakpoint "B" foi acionado ele reabilita o grupo X ou Y. Essa é uma grande vantagem ao se definir grupos aos breakpoints. Para testar esse recurso, adicione ao projeto um novo botão e implemente seu método de click conforme a **Listagem 7**.

O código da **Listagem 7** é bem simples: ele altera o *caption* do botão para "Habilitar Grupo" se o *caption* for "Desabilitar Grupo", e vice-versa. Então, o que será feito é habilitar o grupo de breakpoints "Loop" ao clicar no botão com o *caption* "Habilitar Grupo", e, ao clicar no botão com o outro *caption*, o grupo "Loop" será desabilitado. Para isso, adicione um Breakpoint na linha 4, configurado conforme a **Tabela 4**, e outro na linha 6, que deve ser configurado como a **Tabela 5**.

Listagem 7. Breakpoint que habilita e desabilita grupos

```
01 procedure TForm1.btnHabiltaGrupoClick(Sender: TObject);
02 begin
03   if (btnHabiltaGrupo.Caption = 'Habilitar Grupo') then
04     btnHabiltaGrupo.Caption := 'Desabilitar Grupo'
05   else
06     btnHabiltaGrupo.Caption := 'Habilitar Grupo'
07 end;
08
```

Propriedade	Valor
Condition	
Group	ControlGroup
Break	false
Log message	Habilitando Breakpoint de loop
Enable group	Loop

Tabela 4. Definição do breakpoint para habilitar o grupo Loop

Propriedade	Valor
Condition	
Group	ControlGroup
Break	false
Log message	Desabilitando Breakpoint de loop
Disable group	Loop

Tabela 5. Definição do breakpoint para desabilitar o grupo Loop

Execute a aplicação e, ao clicar no botão "Habilitar Grupo", perceba que além da alteração do *caption* ocorre a criação de um novo item no Log event, porém, como nosso breakpoint já se encontra habilitado, não surtirá efeito algum. Portanto, ao clicar no botão de "Loop", serão disparadas as mensagens ao Log events como já acontecia.

Ao dar um segundo clique sobre o botão, agora chamado "Desabilitar Grupo", perceba na Breakpoint List que o Breakpoint que estava contido no grupo de "Loop" se encontra agora desabilitado. Ao pressionar o botão de "Loop", não serão mais criados itens no Log events.

Debugando Threads

Debugar threads nem sempre é uma tarefa fácil, pois geralmente ocorrem diversos problemas, como o não conhecimento de qual thread está executando. Um outro problema comum é que o debug pode cair em um método de uma thread diferente da thread alvo da nossa depuração. A seguir, um exemplo será apresentado para demonstrar alguns recursos.

O primeiro passo do nosso exemplo é criar uma nova unit e a nomeá-la como *uThreadDebug*. Depois, declare uma classe *ThreadDebug* herdando de *TThread*, e em seu método *execute* implemente o código da **Listagem 8**.

A thread implementada tem um processamento infinito, onde é acrescentado, a cada iteração do loop, o valor de *i* à variável *value*, que é iniciada com o valor do id da thread em cada novo loop do while. Na unit principal adicione a unit *uThreadDebug* a uses e, depois, adicione um *TMemo* e um novo botão ao form principal. No novo botão adicionado implemente seu evento *onClick* como mostra a **Listagem 9**.

Listagem 8. Método execute da Thread

```
01 procedure threadDebug.Execute;
02 var i, j, value : integer;
03 begin
04  FreeOnTerminate := true;
05  while not Terminated do
06  begin
07   try
08    value := ThreadID;
09    for i := 0 to 10 do
10     for j := 0 to 10 do
11      value := value + i;
12   except
13    on E:Exception do
14   end;
15  end;
16 end;
```

Listagem 9. Implementação do botão btnExecThread

```
01 procedure TForm1.btnExecThreadClick(Sender: TObject);
02 var
03  objThreadDebug : threadDebug;
04 begin
05  objThreadDebug := threadDebug.Create();
06  mmo.Lines.Add(IntToStr(objThreadDebug.ThreadID));
07 end;
```

A cada clique no botão *btnExecThread* uma nova instância de *ThreadDebug* será criada e o valor de sua propriedade *ThreadID* será adicionado ao *TMemo* a fim de facilitar a compreensão e o acompanhamento das próximas etapas. Adicione um debug a linha 8 da **Listagem 8** e execute a aplicação.

Clique duas vezes ou mais sobre o botão *btnExecThread* e perceba que, ao cair no breakpoint e pressionar F8, não se sabe a qual thread pertence aquele bloco. Uma forma de resolver esse problema é com a utilização do *ThreadID* para definir uma condição no breakpoint. Isso deverá ser feito em tempo de execução após a instância da classe, onde poderá ser pego o valor da *ThreadID*.

Note, porém, que esse valor irá mudar a cada nova instância, então a cada teste o valor da *Condition* do Breakpoint deve ser alterado para o novo *ThreadID*. Ter que informar uma *Condition* não é um problema, mas existem recursos melhores para isso: um deles é o uso do campo *Thread* da janela Breakpoint Properties. Esse combobox é carregado com as IDs utilizadas e as IDs de threads em execução, facilitando assim a alteração da thread a ser debugada e fazendo com que não seja necessário ficar capturando o *ThreadID*. Temos também o já apresentado Thread Status.

Caso não se saiba nada sobre o problema enfrentado com as threads, é indicado que se pause todas elas e as execute uma por vez, como se estivesse rodando uma aplicação mono thread. Isso facilitará a análise. Se o problema estiver relacionado a threads concorrentes, por exemplo, isso impedirá que uma thread seja executada junto com outra, o que poderia atrapalhar na análise, pois não seria mais possível simular o problema. Na maioria dos casos, o ideal é que se pause as demais threads, ou ao menos as de mesma família, já que podem estar relacionadas.

Debugar uma aplicação é uma tarefa muitas vezes complicada, mas melhora com o tempo e prática. Não existe uma fórmula mágica de como ou por onde começar, depende muito do seu conhecimento sobre aplicação e as ferramentas de trabalho, porém uma boa dica é começar por uma boa linha de raciocínio partindo de pontos onde se domina.

O primeiro ponto é sempre simular o problema: se o problema é simulável, já fica mais fácil corrigi-lo, pois, ao reproduzi-lo, podemos observar quando os erros ocorrem e encontrar mais facilmente uma solução (corrigir problemas que não foram simulados é um verdadeiro "Chute no escuro").

O segundo passo é tentar identificar o macro bloco do problema, e em alguns casos fazer a redução do escopo, eliminando pontos que não afetam o problema. Reduzir não quer dizer que se deva remover blocos de código. Isso pode acontecer, mas com recursos como o New EIP e Log pode-se reduzir o escopo apenas com saltos para locais estratégicos ou geração de bons log.

Para boas análises, aprenda mais sobre os métodos e classes que estão sendo utilizado, o que cada uma faz e como fazem, verifique sempre nomes de variáveis e seus valores, busque por pontos com problemas de legibilidade e códigos mal escritos, utilize sempre códigos limpos para facilitar no debug e nomeie variáveis sempre com nomes fáceis de identificar.

Redundância de código é um dos vilões em problemas que ocorrem em alguns pontos e em outros não. Portanto, verifique se existe mais de um método fazendo a mesma tarefa, identifique loops desnecessários, etc. Depurar código não está somente ligado à correção de erros, também podem ocorrer depurações por prevenção de erros, por isso, sempre que debugar, ao identificar uma área com problema, fique alerta e anote para uma futura correção. Às vezes outros problemas vão estar relacionados a essas áreas, e com tudo mapeado fica mais simples achá-los.

Fique sempre atento a funções que são chamadas seguidas vezes em pontos diferentes, pois elas podem e vão ocasionar problemas de performance.

Para problemas de desempenho, aprenda e aplique ferramentas de métricas no código - elas vão identificar possíveis causas dos problemas, baseadas na complexidade dos algoritmos, acesso ao banco de dados etc.

Em suma, utilize mais dos recursos disponíveis na ferramenta para te auxiliar em tarefas complexas, e até mesmo nas simples. Ademais, evite a utilização de *ShowMessage* no código para verificar valores, como visto, existem recursos de log que, se bem trabalhados, podem gerar logs bem completos do passo a passo da aplicação.

As ferramentas vistas devem ser estudadas mais afundo, pois quanto maior o seu conhecimento sobre elas maior será a facilidade de debugar ou aplicar uma técnica na hora certa. Além de todos os recursos vistos, o Delphi conta com recursos de depuração remota, que são muito úteis em casos em que o problema é simulado apenas no cliente.

Autor

Gutierry Antonio
gutierrydsn@hotmail.com
Entusiasta de Big Data e Data Warehouses, atua como Engenheiro de Software e DBA MySQL. É graduado em Sistemas de Informação e Análise de Sistemas, e atualmente cursa especialização em Engenharia de Sistemas. Possui experiência em desenvolvimento com Delphi, Ruby on Rails e com bancos de dados Firebird e MySQL, além de possuir conhecimento em Java, C#, C++ e nos bancos de dados SQL Server e Oracle, bem como em AWS.

Links:

Embarcadero – Event Log options
http://docwiki.embarcadero.com/RADStudio/Seattle/en/Event_Log_Options

Embarcadero – Event Log Window
http://docwiki.embarcadero.com/RADStudio/Seattle/en/Event_Log_Window

Embarcadero – Using Non-Breaking Breakpoints in Delphi
http://edn.embarcadero.com/article/31263

Desenvolva aplicações com MongoDB no Delphi

Aprenda como persistir os dados da sua aplicação no banco NoSQL MongoDB

Atualmente, com o amplo crescimento do acesso à tecnologia da informação, cada vez mais usuários precisam armazenar e visualizar dados de maneira rápida e eficaz. Entretanto, em alguns casos, é possível notar que a maneira pela qual as informações são armazenadas acaba sendo ineficaz, não atendendo às expectativas e necessidades tanto dos desenvolvedores e demais profissionais de tecnologia, quanto dos usuários, que estão cada vez mais exigentes em relação a soluções rápidas e dinâmicas.

A arquitetura tradicional utilizada na criação de bancos de dados, conhecida como Modelo Relacional, em certos casos pode apresentar certa dificuldade para suportar a quantidade exorbitante de dados que são gerados e acessados hoje em dia, bem como seus formatos variados. Para contornar essa limitação, surgiram os Bancos de Dados NoSQL, alternativas que visam oferecer maior flexibilidade e desempenho diante do aumento do volume e da variedade dos dados.

Fique por Dentro

Este artigo é útil para os desenvolvedores que estão buscando conhecer um pouco mais sobre o paradigma de bancos de dados NoSQL e como utilizá-lo em aplicações Delphi. Aqui serão apresentados os principais conceitos do NoSQL, seus pontos positivos e negativos, e as diferenças existentes quando comparados aos bancos de dados relacionais, de forma a auxiliar o leitor na decisão por utilizar ou não esse tipo de solução. Por fim, veremos como acessar um banco de dados MongoDB no Delphi 10.1 Berlin, utilizando para isso seus componentes e drivers de conexão específicos.

Introdução aos bancos de dados NoSQL

Os bancos NoSQL (acrônimo de *Not Only SQL*) consistem de sistemas de armazenamento de dados que não seguem a estrutura convencional, ou seja, o modelo relacional. Por esse motivo, eles também são chamados de não relacionais.

Bases de dados relacionais são totalmente orientadas a tabelas, as quais possuem uma estrutura fixa, baseada em linhas e colunas. Apesar de ter atendido a diversas demandas por anos, esse modelo acabou se mostrando ineficiente em situações onde há a necessidade de armazenar dados em formatos variados (desestruturados ou semiestruturados). Essa necessidade foi um dos fatores que motivou o surgimento dos bancos NoSQL.

Outra questão crítica, que contribuiu para a consolidação desse novo paradigma, diz respeito à performance dos bancos relacionais ao trabalhar com grandes massas de dados, principalmente quando há muitos relacionamentos entre eles. Isso, aliado às crescentes demandas por escalabilidade e custo reduzido, fez com que soluções de NoSQL fossem criadas exatamente para suprir as necessidades dos mais variados cenários.

Características fundamentais do NoSQL

Um banco de dados NoSQL deve possuir algumas características peculiares e fundamentais que o definem como tal:

• **Escalabilidade horizontal:** o termo escalabilidade diz respeito à capacidade que um sistema possui de suportar o aumento do volume de dados, elevando também seu desempenho. Em bancos de dados relacionais, utiliza-se a escalabilidade vertical, a qual

tem o objetivo de obter um maior ganho de processamento e armazenamento do servidor por meio do upgrade de hardware, como memória, disco rígido ou processadores. Porém, os bancos NoSQL devem suportar a escalabilidade horizontal, que consiste no aumento do número de nós (máquinas) disponíveis para realizar um processamento distribuído. Dessa forma, quando uma operação é executada no banco, o processo é dividido entre vários nós existentes, assegurando assim que não haja queda no desempenho;

• **Replicação de forma nativa:** essa característica contribuirá para a obtenção de maior potencial de escalabilidade e facilitará a recuperação dos dados em tempo reduzido;

• **Acesso aos dados facilitados por APIs simplificadas:** acessar informações de maneira rápida e eficiente, mantendo a simplicidade e facilidade, é um dos principais objetivos dos bancos NoSQL. Isso normalmente é alcançado através de APIs que tornam o acesso aos dados mais fácil;

• **Flexibilidade ou ausência de esquema:** a estrutura dos bancos de dados NoSQL apresenta um esquema flexível ou, em alguns casos, chega a não apresentar qualquer esquema que a defina. Essa característica permite que se trabalhe com dados nos mais variados formatos, entretanto, torna menor a garantia de integridade;

• **A consistência nem sempre é mantida:** bancos de dados NoSQL têm a particularidade de eventualmente não conseguirem manter os dados totalmente consistentes. Essa característica é baseada no Teorema CAP (veja o **BOX 1**).

Diferenças básicas entre bancos relacionais e não relacionais

Após conhecer as principais características do NoSQL, é importante que se possa identificar as diferenças existentes entre os bancos de dados relacionais e os não relacionais (NoSQL). Na **Tabela 1** é possível visualizar as mais marcantes, que podem ser facilmente distinguidas entre esses dois modelos.

BOX 1. Teorema CAP

O Teorema CAP, também conhecido como Teorema de Brewer, foi criado após a afirmação do cientista da computação Eric Brewer de que é impossível um sistema distribuído garantir de forma simultânea três requisitos: Consistência, Disponibilidade e Tolerância ao particionamento (daí a sigla CAP: **C**onsistency, **A**vailability, **P**artition tolerance).

A consistência se refere à situação que ocorre quando um dado é modificado e todos os usuários têm acesso ao mesmo dado atualizado e em tempo real.

A disponibilidade, por sua vez, é a garantia de que o sistema permanece ativo em um determinado período, de modo que cada solicitação receba a confirmação da existência ou não de determinada informação.

Já a tolerância ao particionamento consiste na capacidade que o sistema deve ter de continuar operando mesmo após ocorrer uma falha na rede, que possa tornar algum nó inacessível.

Portanto, segundo o Teorema CAP, o sistema que garante dois desses requisitos fatalmente não disponibilizará um terceiro.

BOX 2. ACID e BASE

O termo ACID se refere a um conjunto de propriedades que os bancos de dados relacionais devem apresentar para garantir a integridade de suas transações: Atomicidade, Consistência, Isolamento e Durabilidade. O objetivo do modelo ACID é tornar o banco de dados totalmente consistente.

Já o acrônimo BASE (**B**asically **A**vailable, **S**oft-State, **E**ventually Consistent), utilizado em bancos de dados NoSQL, indica que o banco deve apresentar como características: ser basicamente disponível, ter estado leve e estar eventualmente consistente. Ou seja, o foco aqui está na disponibilidade e apenas se volta para a consistência em um momento oportuno, garantindo mais simplicidade e rapidez.

Quando usar o NoSQL

Da mesma forma que o modelo relacional apresenta suas limitações, o NoSQL também possui alguns pontos nos quais ele não irá suprir a necessidade do desenvolvedor. Conhecendo melhor suas características e confrontando-as com as particularidades dos bancos relacionais, é possível identificar seus pontos positivos e negativos, habilitando o desenvolvedor a decidir sobre qual modelo é o mais adequado para determinado projeto.

	Banco de dados relacional	Banco de dados NoSQL
Modelagem dos dados (esquema)	Tabelas (colunas, índices, relacionamentos)	Ausente ou flexível (chave hash para recuperar valores)
Consistência	Regras de consistência fortes e rigorosas.	Realizada eventualmente; não garante total consistência dos dados.
Desempenho	Alcançado através da otimização de consultas e criação de índices.	Tamanho do cluster do hardware e latência da rede e da aplicação.
Escalabilidade (tipo mais viável)	Vertical (otimização do hardware)	Horizontal (clusters distribuídos para aumentar a transferência sem aumentar a latência).
Custo da escalabilidade	Alto (custo maior para upgrade em hardware mais robusto)	Relativamente baixo (possível utilizar hardware de baixo custo)
APIs	Operações executadas pelos SGDBs através de comandos SQL.	Estruturas de dados trabalham em memória; busca por chaves hash.
ACID	Totalmente compatível com as propriedades ACID (**BOX 2**).	Substitui algumas propriedades ACID por um modelo mais flexível, o BASE (**BOX 2**).
Disponibilidade dos dados	Não suporta bem uma grande demanda de dados, por não ser eficiente na sua distribuição.	Atende ao grande fluxo de solicitações com alta distribuição e maior disponibilidade.

Tabela 1. Diferenças entre bancos relacionais e NoSQL

Se o sistema necessitar que seja garantida consistência dos seus dados, por exemplo, o NoSQL não é a opção mais adequada, por não ter um controle rígido de transações baseado nas propriedades ACID. Assim, quando a integridade dos dados for indispensável, o modelo relacional deve ser utilizado.

Por outro lado, se o sistema precisa trabalhar com um volume de dados muito grande, que requer flexibilidade e alta escalabilidade para obter um bom desempenho, sem se preocupar com uma total consistência, o NoSQL se torna a solução ideal.

Bancos NoSQL mais utilizados atualmente

Com o crescimento da sua participação no mercado, é cada vez maior o número de grandes empresas que já são adeptas à utilização dos bancos de dados NoSQL em seus processos. Entre elas podemos destacar algumas, como Google, Facebook, Amazon, Ebay, Twitter, LinkedIn e várias outras empresas com atuação no mundo inteiro.

Analisando um panorama mais recente, diversos bancos de dados NoSQL já podem ser considerados ferramentas consolidadas, e muitos deles já estão sendo suportados nas mais diferentes plataformas de desenvolvimento. Entre os que mais se destacam atualmente estão: MongoDB, MarkLogic Server, DynamoDB, CouchDB, Cassandra, Neo4j e Redis.

Classificação dos bancos NoSQL

Os bancos de dados NoSQL podem ser divididos quanto à maneira como os dados são tratados e armazenados. Essa classificação é feita a partir dos modelos ou tipos detalhados a seguir:

• **Key-Value (Chave-Valor):** O modelo Key-Value é o que apresenta a implementação mais simples entre os modelos NoSQL, baseado na estrutura de dados conhecida como tabela hash, na qual cada item é representado por uma chave única com um valor associado.

Nesse modelo a consulta é exclusivamente feita através da chave, por isso não é eficiente em operações mais complexas. Por conseguir suportar uma maior carga de dados e obter um alto desempenho, essa opção é bastante utilizada para caches.

Os principais exemplos de bancos Key-Value são o Redis, o Tokyo Cabinet e o Project Voldemort.

• **Column Family (Família de colunas):** O modelo Column Family, que possui um nível mais alto de complexidade se comparado ao Key-Value, também consiste de um par de chave-valor, porém a chave aponta para múltiplas colunas (família de colunas). Essas famílias, por sua vez, consistem de linhas e colunas identificadas por chaves e um valor timestamp, que distingue versões diferentes do mesmo dado.

Esse modelo permite o particionamento dos dados, com a capacidade de executar pesquisas rápidas e obter forte consistência, sendo melhor aplicado em sistemas distribuídos. Como ponto fraco, pode-se destacar sua API de baixo nível.

Alguns bancos que se enquadram nesse modelo são o BigTable, o Cassandra e o HBase.

• **Documents (Orientado a documentos):** Bancos de dados desse modelo consistem, basicamente, no armazenamento de coleções de objetos que possuem uma chave única associada a um conjunto de campos e seus valores. Esses objetos são chamados de documentos e podem, inclusive, possuir outras estruturas semelhantes aninhadas na forma de atributos.

O modelo orientado a documentos é muito utilizado em aplicações Web e não possui esquema definido, sendo totalmente flexível no sentido de permitir a alteração da estrutura de um documento sem que outros sejam modificados. Assim, dentro de uma mesma coleção (que se assemelha às tabelas nos bancos relacionais) pode haver documentos (semelhantes às tuplas/linhas) com estruturas diferentes.

A base desse modelo são documentos XML e JSON, e suas consultas podem ser executadas sobre a sua chave única ou qualquer outro campo ou registro. Os destaques desse modelo são sua enorme flexibilidade, simplicidade e boa performance, porém ele não garante a consistência absoluta dos dados.

Alguns exemplos de bancos de dados NoSQL orientados a documentos são: MongoDB, CouchDB e MarkLogic Server.

• **Graphs (Orientado a grafos):** Esse é o modelo de maior complexidade, devido à estrutura de dados na qual se baseia: os grafos. Esse formato consiste de duas entidades básicas: os nós, que correspondem aos vértices do grafo, e os relacionamentos, que são as suas arestas. Os vértices são interligados pelas arestas, identificando os relacionamentos existentes entre os nós, conforme representado na **Figura 1**.

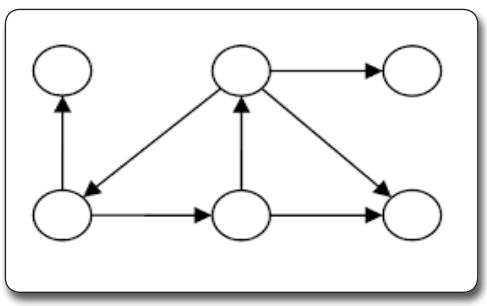

Figura 1. Nós e relacionamentos em um grafo

Os bancos de dados orientados a grafo mais conhecidos são os seguintes: Neo4J, InfoGrid, InfiniteGraph e BrightstarDB.

Introdução ao MongoDB

O MongoDB é um banco de dados NoSQL orientado a documentos, e uma das soluções desse tipo mais difundidas do mercado. Ele foi criado em 2007, pela empresa 10Gen, chamada hoje de Mongo Inc., e é usado atualmente por diversas empresas, como Ebay, EA Sports, Globo.com, FourSquare, LinkedIn e SourceForge.

Detalhes técnicos

Esse sistema de banco de dados foi desenvolvido na linguagem C++ e possui o código-fonte aberto, através da licença GNU AGPL 3.0.

Além disso, trata-se de uma opção multiplataforma, compatível com sistemas operacionais Windows, Linux, Mac OS X e Solaris.

O MongoDB utiliza a sintaxe BSON (uma opção derivada do JSON, mas armazenada em formato binário) para persistência dos dados e já conta com drivers de conexão disponíveis para diversas linguagens e plataformas de desenvolvimento, como Java, C, C++, C#, PHP, Python e Delphi.

A organização dos dados no MongoDB é feita em função da aplicação, ou seja, os dados são organizados de acordo com as necessidades do sistema.

Benefícios do MongoDB

Alguns dos maiores benefícios encontrados no MongoDB são:

- A simplicidade de executar consultas nos dados, pela sua sintaxe de fácil entendimento;
- Trabalhar com dados diretamente na memória RAM, tornando a busca mais rápida e eficiente, resultando em um ótimo desempenho;
- O emprego do conceito de Sharding, que consiste em dividir os dados armazenados em várias máquinas;
- O recurso GridFS, que é capaz de armazenar arquivos diretamente no banco de dados.

Suporte ao MongoDB no Delphi

Entre as novidades apresentadas a partir do Delphi 10 Seattle, está o suporte à conexão com o MongoDB na versão 3.0 ou posterior, que pode ser feita através dos componentes de conexão do FireDAC.

Na versão mais atual do Delphi, a 10.1 Berlin, já foram adicionadas algumas melhorias em relação a esse banco de dados, como o suporte a alertas do banco de dados através de notificações e a inclusão de um novo método para percorrer documentos.

Existem também outros drivers no Delphi que podem ser utilizados para fazer a comunicação com o banco de dados MongoDB, entre os quais podem ser citados o pebongo, que foi o primeiro driver desenvolvido para Delphi, e o TMongoWire.

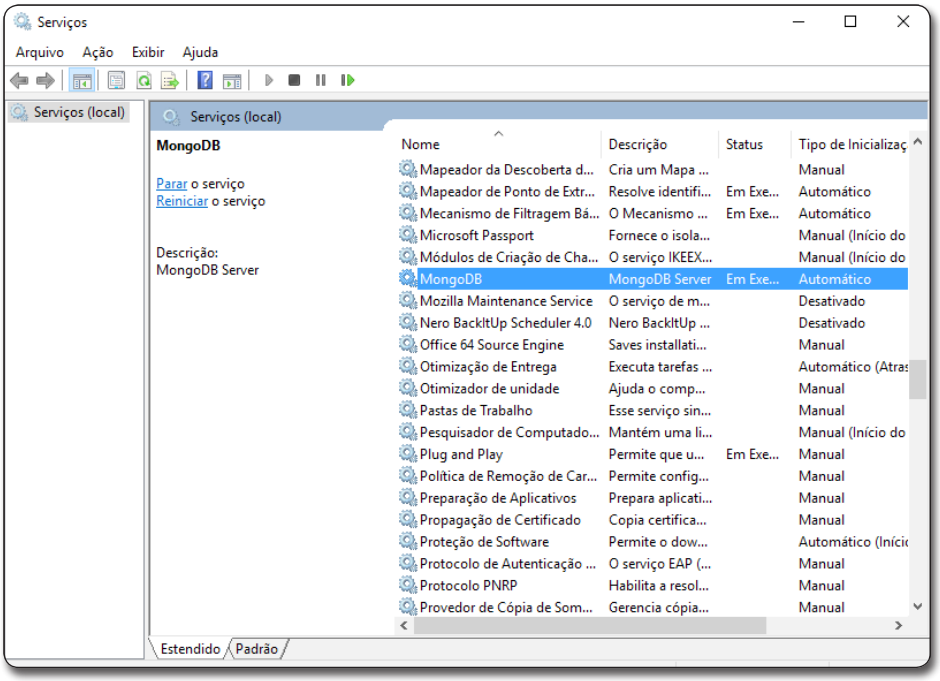

Figura 2. Iniciando o serviço MongoDB no Windows

Instalando o MongoDB

Neste artigo utilizaremos o sistema operacional Windows 10, e para instalar o MongoDB inicialmente é necessário fazer o download de sua versão mais recente, que no momento da escrita desse conteúdo é a 3.2.6. O download pode ser feito diretamente do site do banco de dados, cujo endereço encontra-se na seção **Links**.

Após baixar o pacote de instalação do MongoDB, basta executar o arquivo mongodb-win32-x86_64-2008plus-ssl-3.2.6-signed.msi para iniciar a tela de instalação. A instalação é bem simples, seguindo o modelo *Next* > *Next* > *Install*, o único detalhe é que se deve clicar no botão *Complete* ao invés do *Custom*, para que todas as ferramentas do servidor do MongoDB sejam instaladas. O diretório padrão para instalação do MongoDB é o *C:\Program Files\MongoDB\Server\3.2*.

Depois de instalado, é preciso criar manualmente o diretório *C:\data\db* para que o servidor do MongoDB seja executado. Além disso, as variáveis de ambiente do Windows devem ser configuradas, adicionando o path do servidor da seguinte forma:

- Clicar com botão direito em *Meu Computador* e acessar o menu *Propriedades*;
- Clicar em *Configurações avançadas do sistema*, e na aba *Avançado* clicar no botão *Variáveis de ambiente*;
- Na seção *Variáveis do sistema*, clicar em *Path* e depois em *Editar*;
- Na sequência, ao clicar em *Novo*, deve ser adicionado o diretório do MongoDB: *C:\Program Files\MongoDB\Server\3.2\bin*.

Em seguida, deve ser feita a instalação do MongoDB como um serviço do Windows. Para isso, devemos abrir o prompt de comando como administrador e inserir a seguinte linha:

```
mongod –dbpath=C:\data\db –logpath=C:\data\db\
log.txt –install
```

Com o serviço MongoDB instalado, ainda é preciso iniciá-lo na tela de serviços do Windows, que pode ser acessada através do comando *services.msc* no menu *Executar*. Como mostra **Figura 2,** devemos selecionar o serviço do MongoDB e clicar em *Iniciar serviço* no topo da janela.

Após instalar e configurar o MongoDB, é possível ter acesso ao seu pacote de componentes através do seu console, partindo do prompt de comando do Windows.

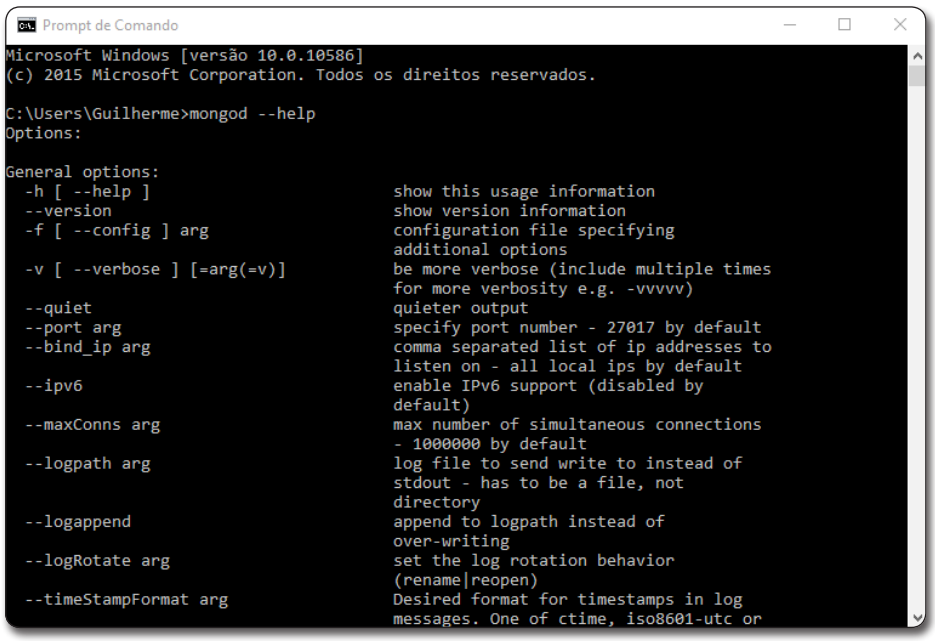

Figura 3. Executando o comando --help do MongoDB no Windows

Listagem 1. Primeiros comandos no MongoDB

```
01 use test
02 db.cadastro.insert({"nome":"José","pais":"Brasil"});
03 db.cadastro.insert({"nome":"Mary","cidade":"Los Angeles","pais":"EUA"});
04 db.cadastro.insert({"nome":"Juan","sobrenome":"Gonzales","pais":"México"});
```

Executando operações através do console

O pacote de componentes do MongoDB apresenta três processos principais, que são: *mongod*, *mongos* e *mongo*.

O *mongod* é o processo central do banco, encarregado de lidar com as solicitações, gerenciar o acesso aos dados e executar as operações em background. Já o processo *mongos* é o controlador e roteador de consulta para clusters distribuídos. Por fim, o processo *mongo* é a interface shell do MongoDB, e fornece um ambiente JavaScript para administradores do sistema e desenvolvedores executarem consultas e operações diretamente no banco de dados.

A **Figura 3** exibe um exemplo de comando executado pelo *mongod*, o "*--help*", que é a função de ajuda do MongoDB e que lista as funções disponibilizadas pelo banco de dados.

Executando o processo *mongo* é possível efetuar diversas operações no banco de dados, como criar uma nova coleção, criar um documento dentro dessa coleção, alterar e adicionar novos campos em um documento específico, entre muitas outras tarefas.

Para utilizá-lo, digite *mongo* no prompt de comandos, e, após a mensagem de boas-vindas, execute os comandos da **Listagem 1**.

A primeira linha faz com que alteremos o banco de dados em uso para um chamado *test*. Observe que mesmo sem ter criado essa base anteriormente, o MongoDB já faz isso para nós e coloca esse banco em uso.

As linhas seguintes inserem novos documentos na coleção *cadastro*, que também não foi criada anteriormente e nem precisou ter um schema definido para receber os registros. Note, ainda, que cada documento tem uma estrutura diferente, e mesmo assim todos foram comportados sem problemas na coleção.

Agora, para listar todos os registros da coleção utilize o comando "*db.cadastro.find();*".

Esses são apenas alguns dos comandos que podem ser utilizados no console. Para ver outros exemplos e funções, visite o site indicado na seção **Links.**

Utilizando o Robomongo

A principal ferramenta utilizada para o acesso aos dados no MongoDB é o Robomongo, que é uma aplicação open source, compatível com as plataformas Linux, Mac OS X e Windows.

O download do instalador da versão Robomongo 0.9.0 (a mais recente no momento da produção deste artigo) pode ser feito através de seu site oficial, disponível na seção **Links**. Sua instalação é rápida e fácil, bastando apenas clicar em *prosseguir* e depois confirmar a instalação, não dependendo de nenhuma configuração adicional.

Através de uma interface simples e amigável (mostrada na **Figura 4**), é possível executar no Robomongo todos os comandos disponibilizados pelo shell do MongoDB, obtendo, dessa forma, uma experiência muito mais agradável e interativa ao realizar consultas e executar outras operações mais complexas no banco de dados.

Na parte superior da tela se encontra o menu principal, que possui opções de conexão, tipos de visualização e configurações específicas do Robomongo. No lado esquerdo da tela é onde está listada a conexão, configurada ao abrir o Robomongo pela primeira vez, e os bancos de dados existentes, com suas respectivas coleções. Já na parte direita da tela pode ser visualizada a área de trabalho do Robomongo, ou seja, é lá que são exibidas as janelas com os documentos pertencentes a uma determinada coleção, além das linhas de código referentes aos comandos executados no MongoDB.

Essa ferramenta, assim como outras semelhantes de gerenciamento de bancos de dados, é importante durante o desenvolvimento e manutenção dos sistemas, uma vez que nos permitirá efetuar as principais operações no MongoDB.

Usando o MongoDB no Delphi

A biblioteca de componentes FireDAC é considerada atualmente como uma das melhores e mais poderosas ferramentas

para conexão com bancos de dados existentes no Delphi. Além de sua fácil utilização, ela oferece alto desempenho no acesso aos dados através de diversas técnicas de otimização, e apresenta drivers de conexão para diversos bancos, como Firebird, MySQL, PostgreSQL, SQL Server, Oracle, SQLite, entre outros.

No caso do MongoDB, além de disponibilizar o driver de conexão, que pode ser selecionado no componente de conexão *FDConnection*, o FireDAC apresenta um conjunto de componentes de acesso específicos para esse banco de dados, agrupados na aba FireDAC NoSQL. Conforme podemos ver na **Figura 5**, esses componentes são: *TFDMongoDataSet*, *TFDMongoQuery* e *TFDMongoPipeline*.

Configurando uma conexão com o MongoDB

Ao iniciar a criação de uma nova aplicação, através do menu *File > New > VCL Forms Application – Delphi*, a conexão com um banco de dados MongoDB deve ser feita com a inclusão do componente *FDConnection* e com a configuração de algumas de suas opções na tela *FireDAC Connection Editor*, de acordo com a **Figura 6.**

O primeiro passo é informar o *Driver ID*, selecionando a opção *Mongo*. Em seguida, no campo *Database*, deve ser inserido o banco de dados utilizado, que nesse caso é *test*, criado anteriormente. Por último, informa-se no campo *Server* o endereço do servidor Mongo, que nesse exemplo será *localhost*.

Após editar as configurações do *FD Connection*, devemos clicar no botão *Test* para testar a conexão com o banco de dados. A tela de login do FireDAC será aberta, e ao pressionar *Ok*, se estiver tudo configurado corretamente, deverá ser exibida a seguinte mensagem: "*Connection established successfully*". Finalizando a configuração do *FDConnection*, deve-se marcar a opção *LoginPrompt* com o valor *False*, para que não seja necessário exibir a tela de login a cada conexão.

Executando operações no banco de dados

Inicialmente, é necessário adicionar na seção *uses*, da seção *interface* da unit, a classe *FireDAC.Phys.MongoDBWrapper*,

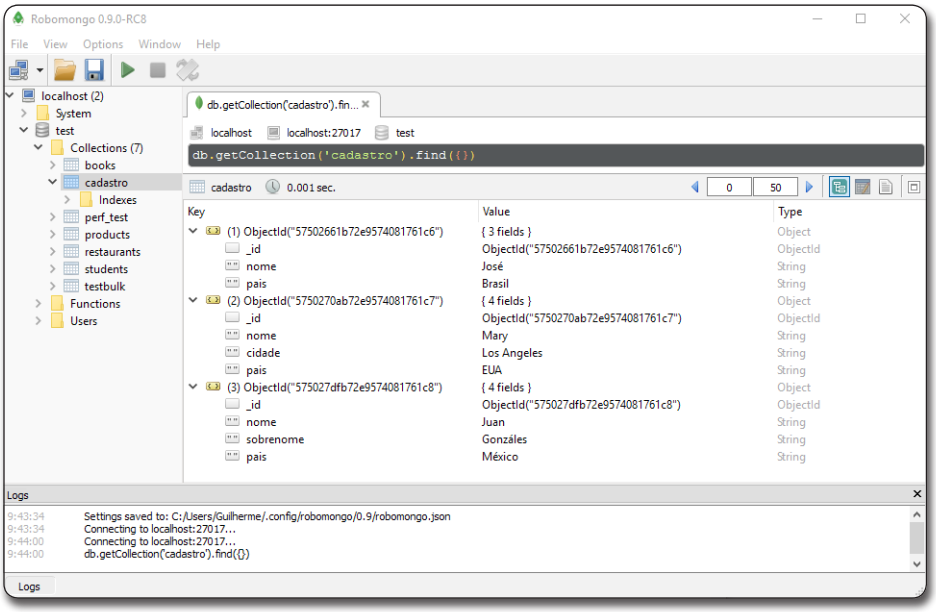

Figura 4. Interface do Robomongo

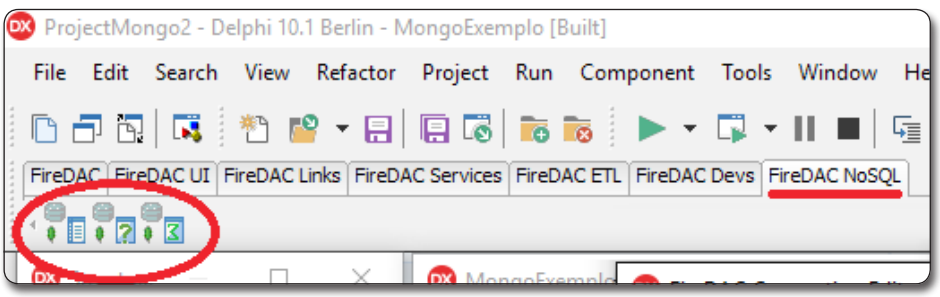

Figura 5. Componentes da aba FireDAC NoSQL.

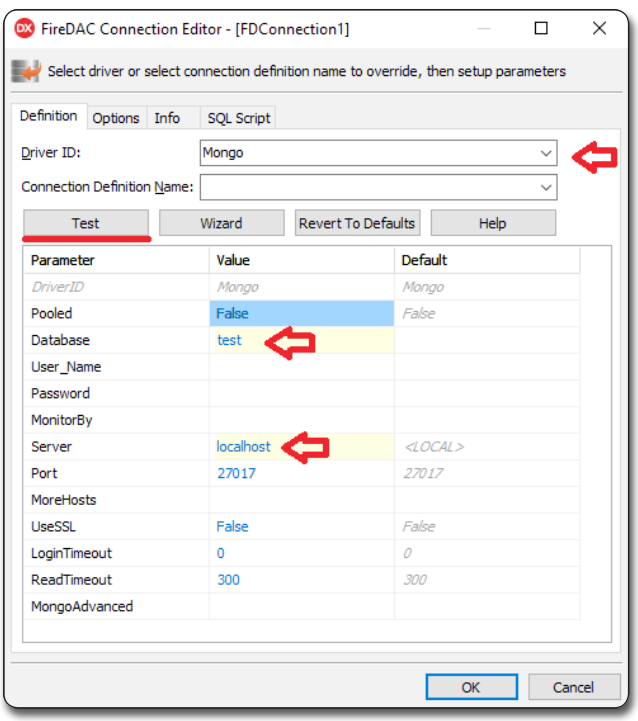

Figura 6. Configuração do FDConnection

para que suas funções possam ser utilizadas no decorrer da implementação. Além disso, também é preciso incluir no projeto uma série de componentes de conexão, os quais são exibidos na **Figura 7** e não necessitam de nenhuma alteração em suas propriedades ou de qualquer outro tipo de configuração.

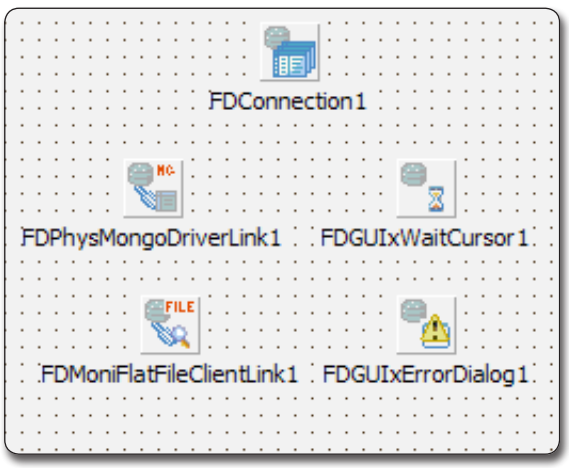

Figura 7. Componentes de conexão do FireDAC

O componente *TFDPhysMongoDriverLink*, localizado na aba *FireDAC Links*, tem como funcionalidade conectar o driver do MongoDB à aplicação, enquanto o componente *TFDGUIxWaitCursor*, encontrado na aba *FireDAC UI*, é usado para controlar o cursor de espera da aplicação. Ambos são de uso obrigatório quando se emprega a biblioteca FireDAC em uma aplicação.

O *TFDMoniFlatFileClientLink*, que também se encontra na aba *FireDAC Links*, e o *TFDGUIxErrorDialog*, localizado em *FireDAC UI*, devem ser usados com objetivos específicos. Enquanto o primeiro é utilizado para trabalhar com arquivos *FlatFile*, ou seja, arquivos que não seguem uma estrutura relacional, o segundo tem a função de exibir exceções do FireDAC na interface do software, através de mensagens detalhadas para o usuário.

Após adicionar os componentes, duas variáveis devem ser declaradas na seção *private*: *FCon*, do tipo *TMongoConnection*, e *FEnv*, do tipo *TMongoEnv*, cujos objetivos são os seguintes:

- *TMongoConnection*: tem como função configurar a conexão para o servidor MongoDB, mantendo acesso aos seus bancos de dados e coleções;
- *TMongoEnv*: seu papel é definir um objeto de serviço que agrupa as referências às bibliotecas do cliente, tratamento de erros, codificadores de conjunto de caracteres, etc.

Essas variáveis são, na verdade, os objetos usados para instanciar as classes indicadas, e através deles será possível acessar os dados do MongoDB. Para isso, é preciso, antes de tudo, conectar a aplicação ao MongoDB no evento *OnCreate* do formulário, conforme mostra a **Listagem 2**.

Nesse procedimento se obtém uma instância da classe *TMongoConnection*, passando como parâmetro a propriedade *CliObj* do *TFDConnection* (linha 4). Antes disso, no entanto, é importante se certificar que a conexão está estabelecida (linha 3). Da mesma forma, deve ser instanciada a classe *TMongoEnv*, através da propriedade *Env* do *TMongoConnection* (linha 5).

Inserindo um documento no MongoDB

Depois de configurar os componentes de conexão, declarar as variáveis, e inicializar os objetos necessário, implementaremos o método *InserirDocumento* no form principal da aplicação, que será utilizado para inserir um documento na coleção *cadastro*. Para isso, empregamos o formato JSON, conforme é exibido na **Listagem 3**.

Listagem 2. Conectar a aplicação ao MongoDB

```
01 procedure TForm1.FormCreate(Sender: TObject);
02 begin
03   FDConnection1.Connected := True;
04   FCon := TMongoConnection(FDConnection1.CliObj);
05   FEnv := FCon.Env;
06 end;
```

Listagem 3. Inserindo um documento no MongoDB

```
01 procedure TForm1.InserirDocumento;
02 var
03   oDoc: TMongoDocument;
04 begin
05   oDoc := FEnv.NewDoc;
06   try
07     // construindo um novo documento
08     oDoc
09       .Add('nome','Gerrard')
10       .Add('sobrenome','Lennon')
11       .BeginObject('endereco')
12         .Add('rua', '3 Avenue')
13         .Add('numero', '1840')
14         .Add('cep', '90084')
15       .EndObject
16       .Add('cidade', 'Liverpool')
17       .Add('pais', 'Inglaterra');
18     // Inserindo-o na colecao
19     FCon['test']['cadastro'].Insert(oDoc);
20   finally
21     oDoc.Free;
22   end;
23 end;
```

A primeira tarefa executada pelo método foi a criação da variável *oDoc*, um objeto do tipo *TMongoDocument* usado para instanciar a classe utilizada para inserção empregando o formato JSON. A classe *TMongoDocument* tem como principal função definir um documento, que é a entidade chave da API do MongoDB, e é responsável por toda a troca de informações, fornecendo diversos métodos e propriedades para lidar com os registros da base.

O objeto *oDoc* foi instanciado através do método *NewDoc*, resultando na criação de um novo documento. Em seguida, foram adicionados os campos existentes no documento, com seus respectivos valores, utilizando os métodos e funções da classe *TMongoDocument*, como *Add*, *BeginObject* e *EndObject*.

Após construir o novo documento, representado pelo objeto *oDoc*, o mesmo foi inserido na coleção *cadastro* por meio da função *Insert*.

Ao finalizar o método, o objeto foi liberado da memória através do comando *free*.

Na **Figura 8** pode ser verificado, ao executar o comando *find()* através do console do MongoDB, que esse novo documento foi inserido com sucesso na coleção.

Pode-se notar que o campo *_id*, que não foi inserido no documento através do código-fonte da aplicação, foi gerado automaticamente pelo MongoDB no formato de chave hash. Caso o desenvolvedor necessite de um ID sequencial, o campo *_id* também suporta a inserção de valores, porém a sua sequencialidade deve ser controlada pela própria aplicação.

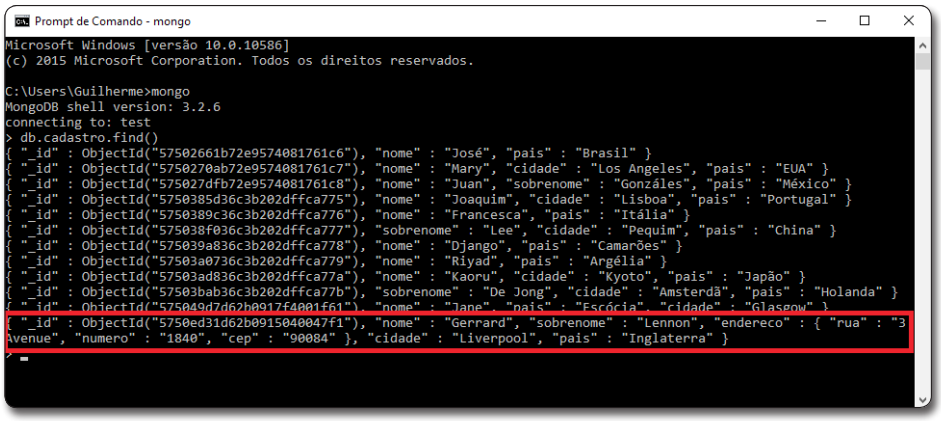

Figura 8. Documento inserido no banco de dados

Atualizando documentos

A alteração/atualização de um determinado documento no MongoDB pode ser necessária por vários motivos, como alterar o valor de um campo, incrementar um valor numérico, inserir um novo campo, remover um campo existente, etc.

O exemplo exibido na **Listagem 4** mostra o método *AlterarDocumentos*, que também deve ser criado no form principal.

Listagem 4. Alterar documento no MongoDB.

```
01 procedure TForm1.AlterarDocumentos;
02 begin
03  // conexão
04  FCon['test']['cadastro'].Update()
05    // critérios
06    .Match
07      .Add('nome', 'José')
08    .&End
09    // alterações
10    .Modify
11      .&Set
12        .Field('sobrenome', 'Silva')
13        .Field('cidade', 'Rio de Janeiro')
14        .Field('idade', 50)
15      .&End
16    .&End
17    .Exec;
18 end;
```

Nesse procedimento o documento é atualizado utilizando o método *Update*, que é dividido em dois blocos: *Match* e *Modify*. No bloco *Match* devem ser definidos os critérios para selecionar o documento que será modificado. O bloco *Modify*, por sua vez, é usado para determinar quais as modificações que serão aplicadas no documento selecionado.

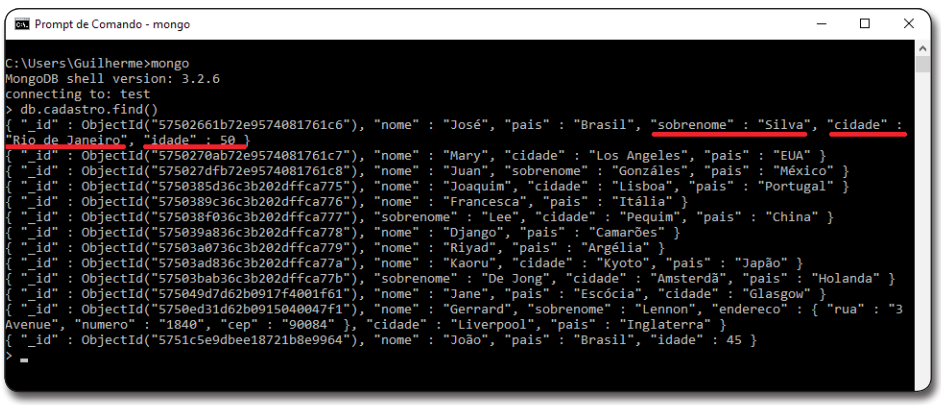

Figura 9. Documento atualizado no banco de dados

Existem diferentes funções que podem ser utilizadas durante a alteração de um documento. O caso mostrado nesse exemplo emprega a função *Set*, que funciona com o objetivo de definir novos valores para seus respectivos campos.

Um ponto importante aqui é que, por meio da função *Set*, também é possível indicar valores para campos que ainda não existem no documento, sem que seja necessário criá-los previamente, como acontece com o campo idade. Essa possibilidade está adequada ao conceito de flexibilidade do NoSQL.

No final desse método, utiliza-se o comando *Exec* para que a operação seja executada diretamente no documento selecionado.

Ao executar o comando *find()* no console do MongoDB, conforme mostrado na **Figura 9**, pode-se confirmar que o documento foi devidamente atualizado, de acordo com os valores que foram passados no método *AlterarDocumentos*.

Removendo documentos da coleção

A remoção de um ou mais documentos de uma coleção pode ser feita usando a função *Remove* do objeto *FCon*. Assim como a função *Update*, a *Remove* utiliza o comando *Match* para aplicar o critério a ser utilizado na operação de remoção, com os parâmetros sendo passados por meio do comando *Add(<campo>,<valor>)*.

A **Listagem 5** mostra o método *RemoverDocumentos*, criado no mesmo form.

Esse procedimento emprega a função *Remove* (linha 3), passando no bloco *Match* (linha 4) a condição para remover apenas os documentos da coleção *cadastro* que tenham o valor do campo *cidade* seja igual a "Glasgow" (linha 5). Por fim, utiliza-se o método *Exec*, na linha 7, para que o comando seja executado no MongoDB.

Para comprovar que o documento foi realmente removido da coleção, pode-se utilizar novamente o console do

MongoDB e executar o comando *find* para buscar documentos cujo atributo *cidade* seja igual a "Glasgow", conforme a linha de código a seguir:

```
db.cadastro.find({"cidade":"Glasgow"})
```

Ao executar esse comando, nenhum documento deverá ser encontrado, como mostra a **Figura 10.**

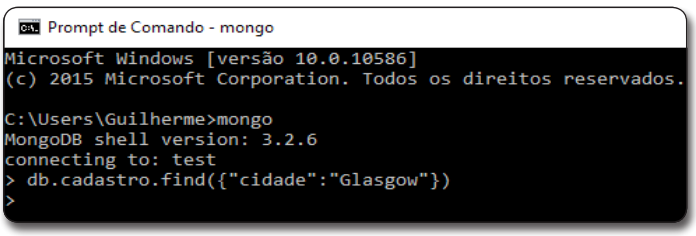

Figura 10. Nenhum documento encontrado

Listagem 5. Remover documento no MongoDB.

```
01 procedure TForm1.RemoverDocumentos;
02 begin
03   FCon['test']['cadastro'].Remove()
04     .Match()
05       .Add('cidade','Glasgow')
06     .&End
07   .Exec;
08 end;
```

Buscando documentos no MongoDB

Assim como as operações de inserção, atualização e remoção de documentos no banco de dados, também é possível executar consultas e exibir as informações na tela, fazendo a comunicação com o servidor do MongoDB por meio de arquivos JSON.

De maneira semelhante às operações apresentadas até agora, desenvolveremos um método para a busca de informações no banco. Nesse caso, o objeto o*Crs*, do tipo *TMongoCursor*, precisa ser criado. Ele representa um cursor do MongoDB, o qual será utilizado para percorrer documentos de uma coleção. A **Listagem 6** exibe o código-fonte utilizado no método *BuscarDocumentos*.

Esse método executa a consulta nos documentos da coleção *cadastro* através da função *Find* e armazena os resultados obtidos no objeto *oCrs*.

Para visualizar os resultados na interface da aplicação, é necessário percorrer todos os documentos do *TMongoCursor* executando o comando de repetição *while*. Em cada iteração do laço, devemos então adicionar o documento atual em algum componente que seja capaz de exibir os dados na tela. Nesse caso, utilizamos um *TMemo* (linha 10). Ao percorrer o conjunto de documentos, cada um deles é adicionado no componente *Memo1* através do método *Lines.Add*.

Ainda dentro do loop é utilizada a função *Count*, na linha 14, para contar o número de documentos que foram encontrados, com seu resultado obtido sendo inserido na propriedade *Caption* do componente *TLabel*.

A **Figura 11** mostra os resultados da busca sendo exibidos no memo.

Listagem 6. Buscar documentos no MongoDB.

```
01 procedure TForm1.BuscarDocumentos;
02 var
03   oCrs: IMongoCursor;
04 begin
05   // encontrar os documentos
06   oCrs := FCon['test']['cadastro'].Find();
07
09   // percorrer o cursor
09   while oCrs.Next do
10     Memo1.Lines.Add(oCrs.Doc.AsJSON);
11
12   // conta total de documentos
13   Label1.Caption :=
14     FCon['test']['cadastro'].Count().Value().ToString()+
15       ' documentos encontrados.';
16 end;
```

Nesse exemplo, a função *Find* foi empregada sem que fosse utilizado nenhum critério de busca, retornando assim todos os documentos da coleção *cadastro*. No entanto, por meio dos comandos *Match* e *Sort* é possível que sejam indicados critérios de busca e ordenação da consulta, respectivamente.

O trecho de código exibido na **Listagem 7** mostra como especificar o filtro da busca e a ordenação dos dados encontrados.

No bloco *Match*, que já foi visto anteriormente no método *AlterarDocumentos*, são definidos os critérios para a busca no banco de dados. Nesse caso, está sendo utilizado o critério para filtrar somente os documentos cujo campo *pais* é igual a "Brasil".

O bloco *Sort* tem a função de indicar qual tipo de ordenação deve ser utilizada e a partir de qual campo a ordenação será realizada. No exemplo da **Listagem 7** ela foi feita sobre o campo *idade* em ordem crescente, através do comando *Ascending*.

Listagem 7. Buscar documentos com filtro e ordenação.

```
01 oCrs := FCon['test']['cadastro'].Find()
02   .Match
03     .Add('pais','Brasil')
04   .&End
05   .Sort
06     .Ascending(['idade'])
07   .&End;
```

Ao executar o método usando essas opções de critério e de ordenação, a aplicação deverá exibir no *Memo1* somente os dois registros que possuem o campo *pais* igual a "Brasil", conforme pode ser visto na **Figura 12.**

Acessando dados pelo TFDMongoQuery

É possível buscar, exibir e manipular documentos existentes no MongoDB através de componentes específicos que obtêm acesso

à base de dados, como o *TFDMongoQuery*, disponibilizado pelo pacote de componentes do FireDAC.

O *TFDMongoQuery* é responsável por buscar as informações diretamente no banco de dados e exibi-las na interface da aplicação, com a ajuda de componentes como o *TDataSource* e o *TDBGrid*.

Em primeiro lugar, é necessário inserir no formulário os três componentes citados anteriormente, e depois devem ser feitas algumas configurações nas suas propriedades pela janela *Object Inspector* do Delphi para que o *TDBGrid* possa exibir os documentos que pertencem à coleção *cadastro*.

O controle *FDMongoQuery1* (nome que o *TFDMongoQuery* deve assumir por padrão ao ser adicionado no form) deve ser ligado no *FDConnection1* através da propriedade *Connection*, e em seguida devem ser configurados os campos *DatabaseName*, que é o nome do banco de dados utilizado, e *CollectionName*, que indica o nome da coleção a ser exibida, inserindo os valores: *test* e *cadastro*, respectivamente.

Depois de configurar o *FDMongoQuery1*, basta ligar os demais componentes de modo que o *DataSource1* tenha a sua propriedade *DataSet* apontada para o componente *FDMongoQuery1*, e que no *DBGrid1* a propriedade *DataSource* esteja apontada para o componente *DataSource1*.

A **Listagem 8** exibe o código-fonte do método *BuscarDocumentosQuery*, criado para efetuar a busca dos dados através do *FDMongoQuery1*.

Listagem 8. Buscar os dados no banco com TFDMongoQuery:

```
01 procedure TForm1.BuscarDocumentosQuery;
02 begin
03   with FDMongoQuery1 do
04   begin
05     Close;
06     QMatch := ('"pais":"Brasil"');
07     Open;
08   end;
09 end;
```

A busca das informações no banco de dados usando o *FDMongoQuery1* consiste apenas em abrir a conexão desse componente com o banco de dados, pelo fato de que suas propriedades já foram configuradas para se comunicar com o banco de dados por meio do *FDConnection*.

Sendo assim, basta usar a função *Open* para abrir a conexão do *TFDMongoQuery* com o servidor do MongoDB. Observe, ainda, que no método *BuscarDocumentosQuery*, antes que qualquer operação seja realizada, foi utilizada a função *Close* para garantir que a query não está indevidamente aberta.

Nesse procedimento também é usada a função *QMatch*, que tem como objetivo definir critérios para filtrar os dados da consulta. Essa função recebe um valor do tipo string com as condições que devem ser aplicadas para que os dados sejam encontrados de acordo com os critérios definidos. Após a abertura da query através da função *Open* os dados são exibidos no *DBGrid1*.

A **Figura 13** mostra os dados da consulta exibidos em um TDBGrid, respeitando o critério definido, que é exibir somente os documentos cujo campo *pais* seja igual a "Brasil".

As funções utilizadas para inserir, alterar, remover e buscar documentos no banco de dados MongoDB são apenas algumas das inúmeras funcionalidades que o FireDAC disponibiliza no Delphi.

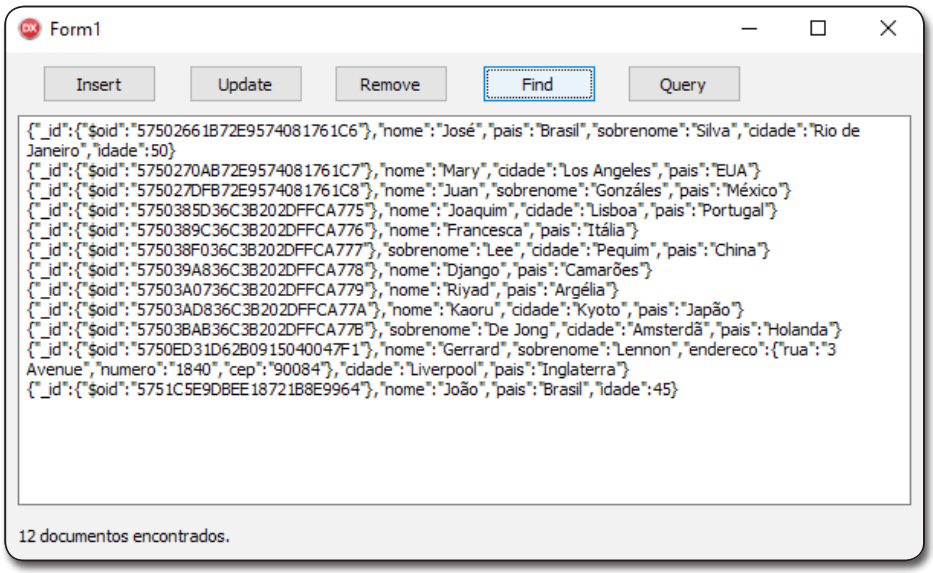

Figura 11. Consulta realizada no banco de dados

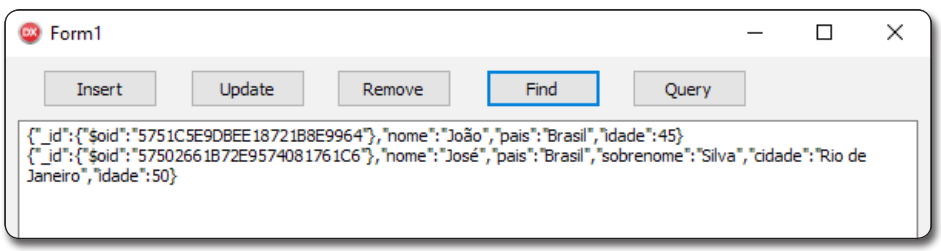

Figura 12. Busca realizada com critério e ordenação

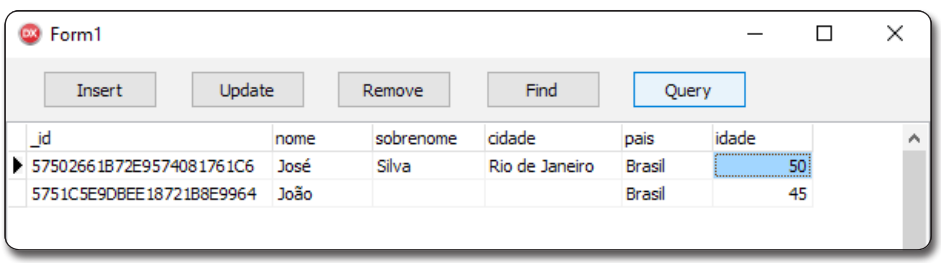

Figura 13. Documentos exibidos no DBGrid1 pelo FDMongoQuery1

Com os componentes, funções e métodos aqui demonstrados por meio de alguns exemplos práticos, torna-se possível alcançar uma boa base inicial para se aprofundar um pouco mais no estudo sobre o MongoDB, aproveitando os ótimos recursos que o Delphi oferece para construir aplicações com o que há de mais atual quando se trata de NoSQL, inclusive no desenvolvimento de aplicações RESTful.

No entanto, após conhecer melhor o conceito de bancos não relacionais e suas principais ferramentas, é fundamental que o desenvolvedor compreenda que o NoSQL não foi criado para substituir de uma vez por todas os bancos de dados relacionais e ser a solução absoluta de todos os problemas.

É necessário, primeiramente, entender suas características, benefícios e limitações antes de decidir se um banco de dados NoSQL é a melhor opção a ser utilizada em um determinado projeto. Somente dessa forma é possível enxergar com clareza como esse conceito pode ser extremamente viável e eficaz em certas situações e em outras nem tanto, para, consequentemente, definir como e quando usar o NoSQL e aproveitar ao máximo tudo que ele proporciona.

Autor

Guilherme Lourenção Tempesta
Bacharel em Ciência da Computação pelo Centro Universitário do Sul de Minas (UNIS-MG) e pós-graduado em Engenharia de Software pela Universidade Gama Filho. Desenvolvedor de software com Delphi e banco de dados Firebird.

Links:

Download do MongoDB
http://www.mongodb.org/downloads

Guia rápido do MongoDB
https://mongodbwise.wordpress.com/2014/05/22/mongodb-guia-rapido/

Download do Robomongo
http://robomongo.org/

MongoDB no Delphi 10.1 Berlin:
http://docwiki.embarcadero.com/RADStudio/Berlin/en/Connect_to_MongoDB_Database_(FireDAC)

Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,
assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486